Lesson 3

Loïc Dutrieux

October 14, 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

During the previous lectures, you saw how to TODO(finish that sentence). Today the emphasis of the lecture is mostly oriented toward carrying a R project. Therefore, some recurrent aspects of project development, such as, how to find help, or techniques to organize your code are covered in the lecture. Scripting means that you often go beyond easy things and therefore face challenges; as a consequence, it is likely that at one moment or another, you will have to go look for help. Part of this lesson is intended to guide you through ways of finding help, including posting to the mail list and writing a reproducible example. In addition, this lecture goes through a couple of essential "good practices" when working on geo-scripting projects. Not all aspects of good practices are covered, however, for those of you who are curious to know more about version control, R package building, etc, be sure to check the reference section of this document.

3 Finding help

There are many place where help can be found on the internet. So in case the function or package documentation is not sufficient for what you are trying to achieve, google is your best friend. Most likely by googling the right key words relating to your "problem", google will direct you to the archive of the R mailing list, or to some discussions on Stack Exchange `http://stackexchange.com/`. These two are reliable sources of information, and it is quite

likely that the problem you were trying to figure out has already been answered before.
However, it may also happen that you discover a "bug" or something that you would qualify of abnormal behavior, or that you really have a question that no-one has ever asked (corollary: has never been answered). In that case, you may submit a question to one of the R mailing list.
For general R question there is a general R mailing list https://stat.ethz.ch/mailman/listinfo/r-help, while the spatial domain has its own mailing list https://stat.ethz.ch/mailman/listinfo/r-sig-geo. Geo related question should be posted to this latter mailing list.
Warning, these mailing list have heavy mail traffic, use your mail client efficiently and set filters, otherwise it will quickly bother you.
These mailing lists have a few rules, and it's important to respect them in order to ensure that:

- 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

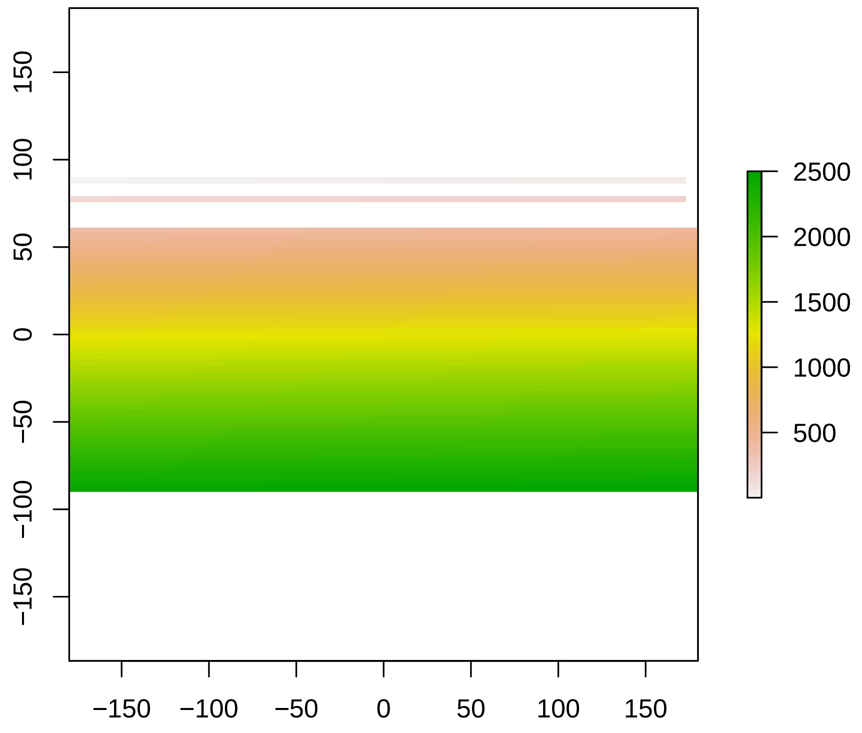
Indispensable when asking a question to the online community, being able to write a reproducible example has many advantages. First it may ensure that when you present a problem, people are able to answer your question without guessing what you are trying to do. BUt reproducible examples are not only to ask questions; they may help you in your thinking, developing or debugging process when writing your own functions. For instance, when developing a function to do a certain type of raster calculation, start by testing it on a small auto-generated rasterLayer object, and not directly on your actual data covering the entire universe ...

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

> plot(r)
```



Useful to know when writing a reproducible example; instead of generating your own small datasets (vectors or RasterLayers, etc) as part of your reproducible example, use some of R ”built-in” datasets. They are part of the main R packages. Some popular datasets are: cars, meuse.grid_ll, Rlogo, iris, etc The autocompletion menu of the data function will give you an overview of the datasets available

```
> # This demonstration of datasets still need to be writen
> data(cars) # Imports the variable cars in the working environment
> class(cars)
```

```
[1] "data.frame"
```

```
> # head(cars) # Visualizes the first five rows of the variable
```

```
> plot(cars) # The plot function on this type of dataset (class = data.frame, 2 column) au
```

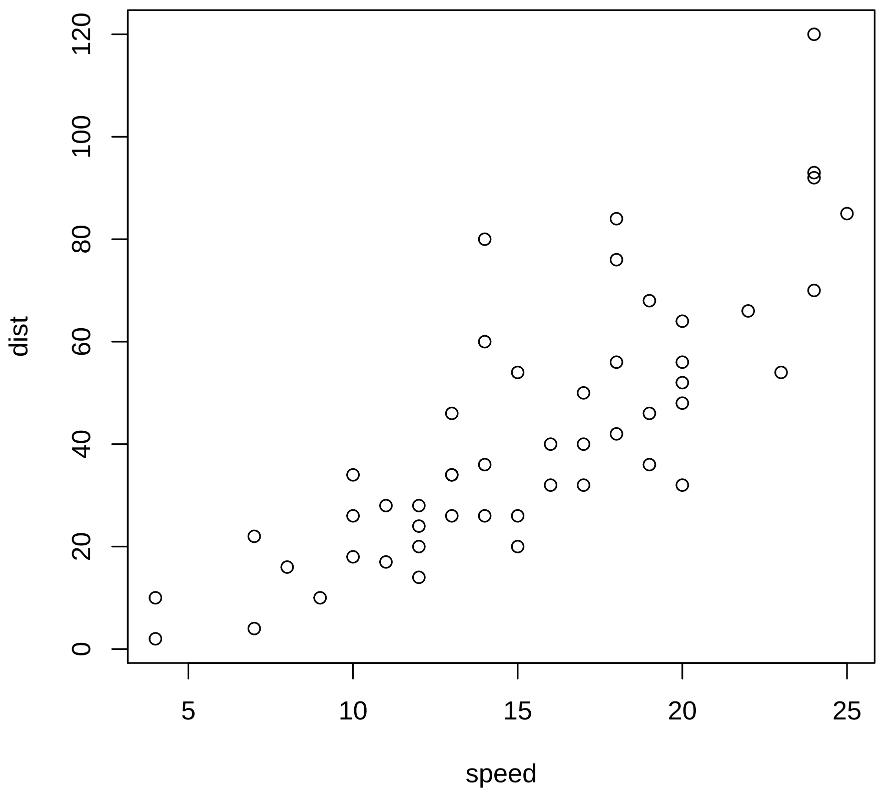


Figure 1: Visual representation of the cars dataset

```
> ## 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
- Follow the R style guide `http://google-styleguide.googlecode.com/svn/trunk/Rguide.xml`
- Document your functions
- Build a package
- Keep a TODO list
- Use version control to develop/maintain your package

Function writing

The objectif of this section is to provide a few advices to help you write efficiently function. That is functions that are simple, generic and flexible, so that they integrate well in a processing/analysis chain and can easily be re-used in a slightly different chain if needed. More flexibility in your function can be achieve thanks to some easy tricks provided by control flow. The following 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

One way of making function more auto-adaptive is by adding checks of the input variables. Using object class can greatly simplify this task. For example let's imagine that you just wrote a simple HelloWorld function.

```
> HelloWorld <- function (x) {
+   hello <- sprintf('Hello %s', x)
+   return(hello)
+ }
> HelloWorld('john')
```

```
[1] "Hello john"
```

Obviously, the user is expected to pass an object of character vector to x. Otherwise the function will return an error. You can make it not crash by controling the class of the input variable. For example.

```
> HelloWorld <- function (x) {
+   if (is.character(x)) {
+     hello <- sprintf('Hello %s', x)
+   } else {
+     hello <- warning('Object of class character expected for x')
+   }
+   return(hello)
+ }
> HelloWorld(21)

[1] "Object of class character expected for x"

>
```

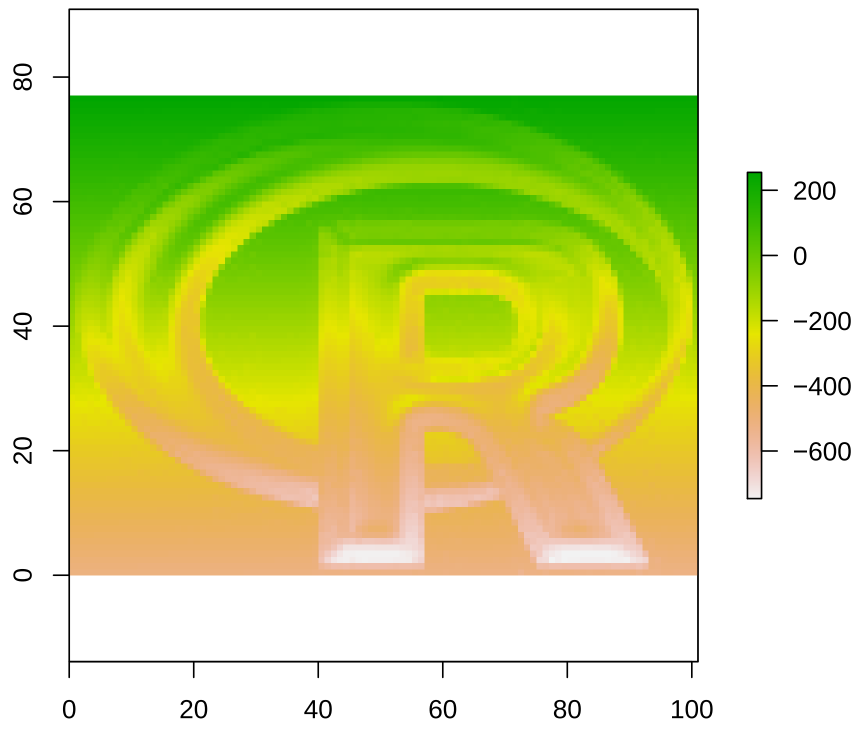
The function does not crashes anymore, but returns a warning instead.

Note that most common object classes have their own logical function, that returns TRUE or FALSE. For example.

```
> is.character('john')

[1] TRUE

> # is equivalent to
> class('john') == 'character'

[1] TRUE

> is.character(32)

[1] FALSE

> is.numeric(32)

[1] TRUE
```

Also note that is.character(32) == TRUE is equivalent to is.character(32). Therefore when checking logical arguments, you don't have to use the == TRUE. As an example, a function may have an argument (plot) that if set to TRUE will generate a plot, while if set to FALSE does not plot. This means that the function certainly contains an if statement. if(plot) in that case is equivalent to if(plot == TRUE), it's just shorter.
An example, with a function that substract 2 RasterLayers, with the option to plot the resulting RasterLayer, or not.

```
> library(raster)
> minusRaster <- function(x, y, plot=FALSE) { # Function to substract 2 rasterLayers
+   z <- x - y
+   if (plot) {
+     plot(z)
+   }
+   return(z)
+ }
> # Let's generate 2 rasters
> r <- raster(system.file("external/rlogo.grd", package="raster")) # that one is the R log
> r2 <- r # The second RasterLayer is calqued from the initial RasterLayer in order to avo
> r2[] <- (1:ncell(r2)) / 10 # Filling the rasterLayer with new values. The /10 simply mak
> r3 <- minusRaster(r, r2) # simply performs the calculation

> r4 <- minusRaster(r, r2, plot=TRUE) # Performs the claculation and plots the resulting R
```



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. So try is a perfect way to deal with heterogeneous/unpredictable input data.
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 that illustrate the use of try for sequencially calculating frequency on a list of auto-generated RasterLAyers.

```
> library(raster)
> # Create a raster layer and fill it with "randomly" generated integer values
> a <- raster(nrow=50, ncol=50)
> a[] <- floor(rnorm(n=ncell(a)))
> # The freq() function returns the frequency of a certain value in a RasterLayer
> freq(a, value=-2)
> # Let's imagine that you want to run this function over a whole list of RasterLayer
> # But some elements of the list are impredictibly corrupted, so the list looks as follow
> b <- a # So that b is also a RasterLayer
> c <- NA
> list <- c(a,b,c)
> # Running freq(c) would return an error and crash the whole process
> out <- list()
> for(i in 1:length(list)) {
+   out[i] <- freq(list[i], value=-2)
+ }
> # So by building a function that includes a try(), we are able to catch the error, allow
> fun <- function(x, value) {
+   tr <- try(freq(x=x, value=value), silent=TRUE)
+   if (class(tr) == 'try-error') {
+     return('This object returned an error')
+   } else {
+     return(tr)
+   }
+ }
> # So let's try to run the loops again
> out <- list()
> for(i in 1:length(list)) {
+   out[i] <- freq(list[i], value=-2)
+ }
> out
> # Note that using a function of the apply family would be a more elegant/shorter way to
> (out <- sapply(X=list, FUN=fun, value=-2))
>
```

Package building

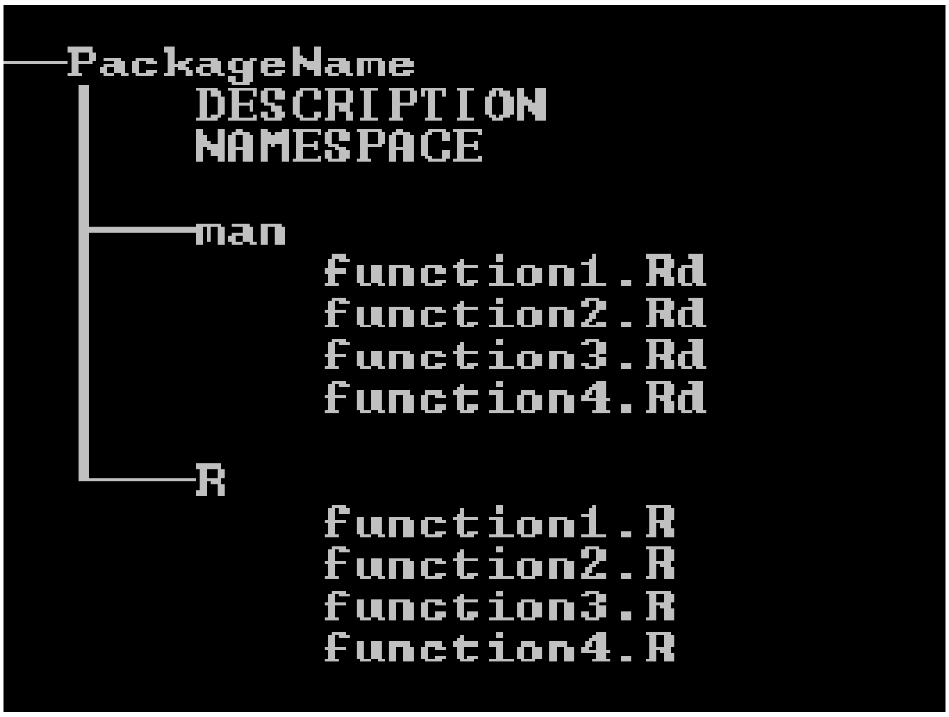
Also part of good programming practices, building a R package is a great way to stay organized, keep track of what you are doing and be able to use it quickly and properly at any time.

Why building a package?

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

For these reasons, if you build a package, next year you will still be able to run the functions you wrote yesterday. 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



```
──PackageName
   │     DESCRIPTION
   │     NAMESPACE
   │
   ├─────man
   │          function1.Rd
   │          function2.Rd
   │          function3.Rd
   │          function4.Rd
   │
   └─────R
              function1.R
              function2.R
              function3.R
              function4.R
```

Figure 2: Minimal structure of an R package

This sections describes the structure of a simple package, for more information about R packages in general and details of package creation, please refer to the manual *Writing R extensions* `http://cran.r-project.org/doc/manuals/R-exts.html`.
The default basic structure of a package is as follows: 2 subdirectories (R and man), NAMESPACE and DESCRIPTION files (Figure 2). 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 directory**. 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.

Useful tips for package creation

The package.skeleton function will help you get the package structure from a list of sourced functions. (A function opened in the R Studio editor pannel can be sourced by pressing the ”Source” in the top right hand corner of the pannel) 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. Explore the menus and option under Project - New project. More package creation tips can be found online `http://mages.github.io/R_package_development/#1`

Package building

- Prerequisites
  In order to be able to build package, you must have **Rtools** installed and properly setup on your system. Refer to the system setup document for more information on how to install Rtools. Once this is done R Studio offers great integration with Rtools, which makes package building really easy. The build pannel in R Studio is by default located in the upper right window.

- Check and build
  In order to be operational on your system the R package you just created has to be checked and built. This is a critical step since everything has to be perfect in order to pass the check step. Simple things like a missing line in a function documentation file mail result in the package not passing the checks. Although a package that does not fully passes the check may still be built in some cases, this is not recommended, and it is therefore preferable to deal with the errors returned in the check console before building.
  Another option for building package is to do so directly from a remote git repository using the devtools package. This aspect of package building, together with general consideration on version control systems is presented in the next section.

Version control

In this lecture, we won’t go into the details of version control, however, we will review some general aspects that make version control a valuable tool for scientists and geo-programmers.

What is version control?

Version control, also called revision control or source control, refers to the management of changes to documents, files or set of files. The version control management can be performed locally or directly on a remote online repository, and the code is usually stored on an online repository, available from anywhere. 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 for version control?

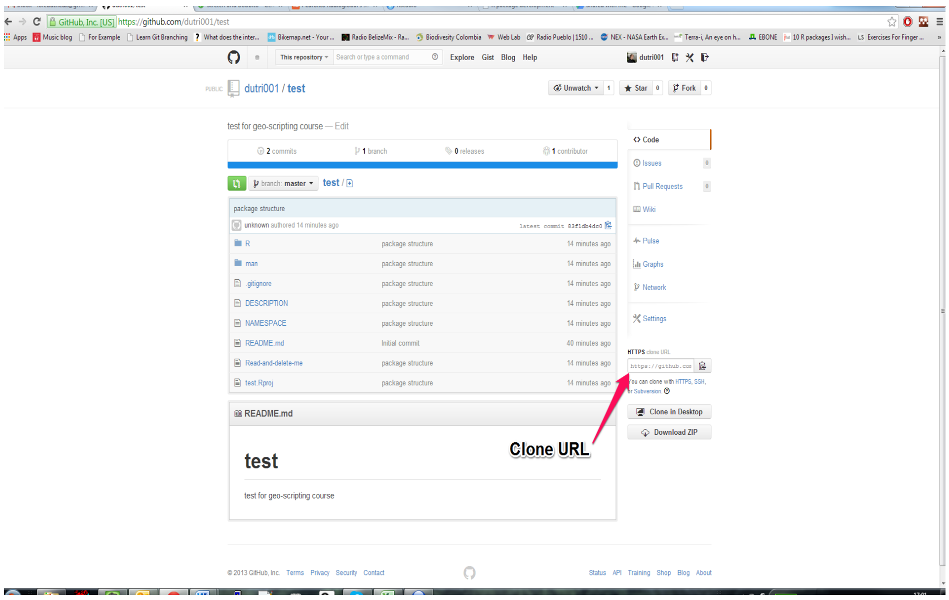
The three main revision control software are Git, Subversion (abreviated as svn), and mercurial (abreviated as hg). Since we are not full time software developpers we are only interested in a limited set of functionalities provided by these systems, most differences between them may appear futile to us. However, a few points of consideration remain, particularly regarding the integration with R Studio and R in general.
Since the R Studio IDE provides integration for git and svn, we will only focus on these two system from here on. Table 1 provides an overview of most online repositories options for svn and git, and level of integration with R package building. Please note that online repositories are not the only options for hosting of the remote repository.

Table 1: Git and svn for R projects

Systems	**Online repositories available**	**Integration with R**
Git	Bitbucket, GitHub, SourceForge	Via devtools package
Svn	R Forge, SourceForge	Via R Forge building tools and install.package() function

Semantics of version control

Commits: Consists of adding the latest changes to the repository. This is done locally with Git and directly on a remote repository with svn. Commits can be seen as a milestones in the versioning of a file or set of files. It is always possible to return to previous commits.
Push (git only): Add previous changes commited to the local repository, to the remote repository.
Clone (git)/**Checkout** (svn): Copy the content of the remote repository locally.
Branch (git): Create a branch (a parallel version of the code in the repository)
Merge: Merges two versions (branches) into one.



test for geo-scripting course — Edit

2 commits · 1 branch · 0 releases · 1 contributor

branch: master · test /

package structure

unknown authored 14 minutes ago · latest commit 83f1db4dc0

R	package structure	14 minutes ago
man	package structure	14 minutes ago
.gitignore	package structure	14 minutes ago
DESCRIPTION	package structure	14 minutes ago
NAMESPACE	package structure	14 minutes ago
README.md	Initial commit	40 minutes ago
Read-and-delete-me	package structure	14 minutes ago
test.Rproj	package structure	14 minutes ago

https://github.com

You can clone with HTTPS, SSH, or Subversion.

README.md

test

test for geo-scripting course

© 2013 GitHub, Inc. Terms Privacy Security Contact Status API Training Shop Blog About

Figure 3: GitHub clone URL

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.

1. Create a GitHub account (only if you don't have one already)

2. From GitHub, create a new repository and name it after the name of the package you want to start

3. Choose for the option to add a README

4. Copy the clone URL of the repository (Figure 3)

5. From R Studio, go to project - Create project - Version Control - Git. In the repository URL field, enter the clone URL that you just copied from GitHub. Project directory name should appear automatically. Choose a location for your project.

6. Copy the content of a package (created using the package.skeleton function) to this newly created directory.

7. In the git panel of R Studio (by default on the upper right), add all the objects added to the package by checking them - press commit - write a commit message - commit - push. You should be prompted for your GitHub username and password, once this is done, your package is added to your GitHub remote repository.

If you are concerned about always having a "compile-able" version of your code, use different branches. Create a "dev" branch on which you develop your new functions or do any kind of experiments, and keep the stable version of your package on the ”master” branch. Once the ”dev” branch becomes stable, merge it with the ”master” branch.