R4DS - Unit 4: R Packages

Andrea Gilardi
March 28, 2023

Outline and main concepts

- The objective of this unit is to present the basic structure of an R package and a set of simple but powerful routines that can be used to build our first package.
- I will showcase the relevant tools and, at the end, we will build an R package that can be used to decide the optimal date for an happy hour.
- Finally, we will synch our package on Github.

What is an R package?

- Following the *R Package - 2e* book, we could say that *An **R package** is the fundamental unit of shareable R code. A package bundles together code, data, documentation, and tests, and is easy to share with others.*

- An R package can be stored on CRAN (Comprehensive R Archive Network), whereas its development version can be stored on Github or other hosting services

- R packages are organised in a standardised format that we must follow. Organising code always makes your life easier since we can follow a template.

What is an R package? (cont)

- A bit of terminology now... A **package** is a directory of files which extend R containing, at minimum, the files DESCRIPTION and NAMESPACE and an R/ directory.
- **A package is not a library**.
- Beware, maintaining and updating an R package can be an extremely time consuming process...

  Maybe I'll become a theoretician. Nobody expects you to maintain a theorem.

Doug Bates (about keeping both Matrix and RcppEigen in sync with CHOLMOD) lme4-author (September 2013)

Peek at the desired product

- Now we are going to develop an R package named `statsAndBooze`. The objective of this package is to find the optimal date for an happy hour given a set of constraint.
- So, for example,

```
library(statsAndBooze)
beer_dates <- parse_dates(
  dates = list(
    andrea = "27-03-2023/1-4-2023" # available from 27/3 to 1/4
    federico = c("29-03-2023", "2-4-2023") # available on 2 days
  )
)
decide_happy_hour(beer_dates)
[1] "2023-03-29"
```

Let's start from scratches...

- During this we class we are going to use the R package `devtools`, so please take a moment to check if it is already installed and, if necessary, install it.

- The following code can be used to generate the skeleton of an empty R package:

```
library(devtools)
create_package("path/for/the/R/package")
```

  So, for example, in a few minutes I'm going to run

```
library(devtools)
create_package("D:/git/statsAndBooze")
```

- The chosen path should point to a non-existing directory that will be created by Rstudio. Do not store an R package inside another R package or a Git repo.

Let's start from scratches... (cont)

- The previous command should open a new Rstudio session that contains the skeleton of an empty R package. We will explore its content in a couple of minutes.
- You should also see a log message like

```
v Creating 'D:/git/statsAndBooze/'
v Setting active project to 'D:/git/statsAndBooze'
v Creating 'R/'
v Writing 'DESCRIPTION'
Package: statsAndBooze
Title: What the Package Does (One Line, Title Case)
Version: 0.0.0.9000
Authors@R (parsed):
* First Last <first.last@example.com> [aut, cre] (YOUR-ORCID-ID)

[truncated...]
```

Now we can analyse the output more precisely together!

Let's start from scratches... (cont)

The following lists the content of the new directory:

- `.gitignore`: The file used to control Git versioning;
- `.Rbuildignore`: Similarly to the `.gitignore` file, this file can be used to exclude some files from the package BUILD.
- `DESCRIPTION`: Stores the metadata of your package (e.g. author, description, dependencies, ...)
- `R/`: The directory where we put the R scripts.
- `NAMESPACE`: declares the functions your package exports and the external functions your package imports from other packages. **DO NOT EDIT BY HAND**.

Check here for more details.

Add Git for Version Control

- Now we started working in a new R session. Therefore, unless you are sharing some `.Rdata` files (

) between sessions, we need to reload `devtools`.
- Then, the `use_git()` function can be used to initialise a Git project inside the repository.
- If you are running in an interactive session, the software might ask you do perform the first commit and restart Rstudio. You should accept the offer.
- As we can see, now there is a Git panel in Rstudio.

Edit the `DESCRIPTION`

Now we can edit the `DESCRIPTION` file:

- `Title: Find the best day to have a beer!`;
- `Version:` I refer you to https://semver.org/;
- `Authors@R`: Just add name, surname, email, and ORCID;
- `License`: I refer you to https://choosealicense.com/;
- `Description:` We are going to fill it at the end.

In case you are developing an R package with some co-authors, you need to list their names (and roles) in the Authors field. See here for more details.

Package dependencies

- As already mentioned, we are trying to develop an R package that parses dates and intervals.
- Working with dates is a nightmare so, instead of defining our own routines, we are going to create wrappers to another package, namely `lubridate`!
- Please notice that when you develop an R package, you cannot use `library()` since that works only for interactive scripting. See also here and here for more details.
- Instead, we can use the function `use_package("pkg")`.

Package dependencies

- You should now see the following output:

```
> use_package("lubridate")
v Adding 'lubridate' to Imports field in DESCRIPTION
* Refer to functions with `lubridate::fun()`
```

  The message highlights that whenever we refer to a lubridate function, we need to add the `lubridate::` prefix. Check also the DESCRIPTION file.

- Please notice that the same behaviour must be applied to any function which is not included into the **base** package.

- **Question:** How can you see which package defines a function?

Interactive development

- Usually, it is much easier if you run the first tests into an interactive session before writing into the package.
- Our first objective is to define a function which takes in input a list of strings and returns the parsed dates:

```
parse_dates(
  dates = list(
    andrea = "2023-03-29", # exactly with this format
    federico = "2023-03-30" # exactly with this format
  )
)
$andrea
[1] "2023-03-29" # NB: it must have class = "Date"

$federico
[1] "2023-03-30"
```

- Now it's your turn! Try to code such a function.

The first function

- Now that we have sketched the skeleton of the function, we can add it to our package. First, we need to create an R script into the R/ folder. We can run `use_R("path.R")`.
- So, for example, I'm going to run

```
use_R("parse.R")
v Setting active project to 'D:/git/statsAndBooze'
* Modify 'R/parse.R'
* Call `use_test()` to create a matching test file
```

- We copy the function definition (**and only the function definition**) into the new script. Every time that we refer to a lubridate function, we need to add `lubridate::`.

The first function (cont)

- The `parse.R` file might look like

```
1. parse_dates <- function(dates) {
2.   lapply(dates, lubridate::as_date)
3. }
4.
```

- Now we can save the file and restart the R session. Then, if you want to make `parse_dates()` available for testing, you can restart the R session and run `load_all()`.
- Let's try it together... NB: delete the function definition and any superfluous library call from the script used for interactive testing.
- If everything works right, now it's a good time for a commit.

R CMD check

- Every time that you modify your R package in a non-negligible way (e.g. you add a new function), you should test that all its moving parts are still working.
- The `R CMD check` command is the gold standard for checking R packages.
- We can run it from the Build panel or via the `check()` function (which is also defined in `devtools`).
- Let's try!

Documentation

- Unfortunately, our new function doesn't have an help file:

```
> devtools::load_all(".")
i Loading statsAndBooze
> ?parse_dates
No documentation for ‘’parse_dates in specified packages and
libraries: you could try ‘??’parse_dates
```

- We can write a specially formatted comment right above the function definition to generate its help page via an R package named `roxygen2`.
- From Rstudio, open `parse.R`, place your cursor somewhere into the function definition and then click on Code -> Insert Roxygen Skeleton.

Documentation (cont)

- You should see something like

```
#' Title
#'
#' @param dates
#'
#' @return
#' @export
#'
#' @examples
parse_dates <- function(dates) {
        lapply(dates, lubridate::as_date)
}
```

- Now we are going to fill all the relevant parts.

Documentation (cont)

- At the end, the output should look like

```
#' Parse a list of strings into dates
#'
#' Please notice that each date must be specified using the YYYY-MM-DD
format.
#'
#' @param dates A list of strings specifying dates. Each entry must be
related to a different person.
#'
#' @return A list with the same length as the input. The strings are
converted into objects of class Date.
#' @export
#'
#' @examples
#' 1 + 1
parse_dates <- function(dates) {
  lapply(dates, lubridate::as_date)
}
```

Documentation (cont)

- We can run `document()` to let `roxygen2` do its magic.
- If you explore the NAMESPACE you should see

```
# Generated by roxygen2: do not edit by hand

export(parse_dates)
```

- Let's run the R CMD check again.
- If everything goes right, this is a good time for another commit!

A minimal R package

- Now we have a minimal working package! We can install it by running `devtools::install()` or using the Build panel.
- After installing our package, we can run the following in a fresh R session

```
library(statsAndBooze)
beer_dates <- list(
  # We can see that our function works with n >= 2 people
  andrea = c("2023-03-29", "2023-03-30"),
  federico = "2023-03-30",
  chiara = "2023-03-30"
)
parse_dates(beer_dates)
```

- If you don't see any error, that means our super simple package works 

To infinity and beyond 🚀

- Now it's time to expand our package! In fact, we said that our objective is to decide a common day for an happy hour.
- For the moment, we just parsed the input constraints into a list of `Date` objects but we are missing the key step: the organization of the happy hour!
- As for the previous case, it's really convenient to start running the first tests into an interactive session.

`decide_happy_hour()` function

- How would you programmatically determine the common day in the following list?

```
library(statsAndBooze)
beer_dates <- list(
  andrea = c("2023-03-29", "2023-03-30"),
  federico = "2023-03-30",
  chiara = "2023-03-30"
)
beer_dates <- parse_dates(beer_dates)
find_happy_hour <- function(beer_dates) {
        ...
}
```

- We need to recursively apply the same function to check which days are shared among pairs of inputs... See the next slide for a solution :)

`decide_happy_hour()` function

- My suggestion would be something like

```
find_happy_hour <- function(dates) {
  lubridate::as_date(Reduce(lubridate::intersect, dates))
}
```

- First, let's see if it works in an interactive session. Then, we can create an ad-hoc `.R` file (e.g. `decide.R`), add the new function and document it.
- **Exercise for home:** Try to understand why we do need the extra call to `lubridate::as_date`.

`decide_happy_hour()` function

- Now, after completing all the previous steps we can load[1] our package and, in a fresh R session, retest that everything works properly:

```
> devtools::load_all(".")
i Loading statsAndBooze
> beer_dates <- list(
    andrea = "2023-03-30",
    federico = "2023-03-30"
  )
> beer_dates <- parse_dates(beer_dates)
> decide_happy_hour(beer_dates)
[1] "2023-03-30"
```

- If everything looks right, let's rerun R CMD check. Please notice that R CMD check automatically re-document our package.

[1] **NB:** load $\neq$ install. See `?devtools::load_all` for more details.

`decide_happy_hour()` function

- **Question:** What is the expected output for the following input?

```
list(
  andrea = "2023-03-29",
  federico = "2023-03-30"
)
```

  Try to formulate an hypothesis and test it running the code.

- Finally, if you don't see any problem, commit again!

Missing steps!

As you can imagine, we just created a super simple package that can be enhanced in many ways. For example:

- `use_readme_rmd()` is a simple routine that can be used to add a README file that documents the main functionalities of our package.
- `use_github()` is another helper function that we can use to link our local Git repository to an online Github session. Please notice that you must not have a Github directory with the same name as the current package.
- **Testing!** We can formulate a series of unit tests to ensure that the functionalities in our package are preserved even if we modified the R code. During the next class we will start from here.

Homework!

TODO :) I will update the slides with the homework this evening.