R4DS - Unit 4: R Packages

Andrea Gilardi
April 5, 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 with 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 happy hour given a set of constraints.

- So, for example,

```r
library(statsAndBooze)
beer_dates <- parse_dates(
  dates = list(
    andrea = "2023-03-27 / 2023-04-01" # available from 27/3 to 1/4
    federico = c("2023-03-29", "2023-04-02") # available on 2 days
  )
)
decide_happy_hour(beer_dates)
[1] "2023-03-29"
```

Let's start from scratches...

- During this 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 named `packageName`:

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

  So, for example, I'm going to run

  ```r
  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 log 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 suspicious 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 to perform the first commit and restart Rstudio. You should accept both offers.
- After the restart, 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... Therefore, instead of defining our own routines, we are going to create wrappers to another package: `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 and see what happened.

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

- **Question:** How can you determine 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

```r
parse_dates <- function(x) {
  lapply(x, lubridate::as_date)
}

```

- 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 x
#'
#' @return
#' @export
#'
#' @examples
parse_dates <- function(x) {
        lapply(x, 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
#'
#' @details Please notice that each date must be specified using the
YYYY-MM-DD format.
#'
#' @param dates A list of strings specifying dates.
#'
#' @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 now 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

```r
library(statsAndBooze)
beer_dates <- list(
  # We can see that our function works with >= 2 people and >= 2 dates
  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 and, currently, we are not doing that...
- In fact, for the moment we are just parsing 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)
list_dates <- list(
  andrea = c("2023-03-29", "2023-03-30"),
  federico = "2023-03-30",
  chiara = "2023-03-30"
)
parsed_dates <- parse_dates(list_dates)
decide_happy_hour <- function(x) {
        ...
}
```

- We need to recursively apply the same function to check which availabilities are shared among different people... See the next slide for a possible solution :)

`decide_happy_hour()` function

- My suggestion would be something like

```r
decide_happy_hour <- function(x) {
  lubridate::as_date(Reduce(lubridate::intersect, x))
}
```

- First, let's see if it works in an interactive session.
- If you don't see any problem, 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
> list_dates <- list(
    andrea = "2023-03-30",
    federico = "2023-03-30"
  )
> parsed_dates <- parse_dates(list_dates)
> decide_happy_hour(parsed_dates)
[1] "2023-03-30"
```

- If everything looks right, rerun R CMD check. Please notice that R CMD check re-documents our package.

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

`decide_happy_hour()` function

- **Question:** What is the expected output of the following code? Please notice that there is no common date.

  ```
  > devtools::load_all(".")
  i Loading statsAndBooze
  > list_dates <- list(
      andrea = "2023-03-29",
      federico = "2023-03-30"
    )
  > parsed_dates <- parse_dates(list_dates)
  > decide_happy_hour(parsed_dates)
  ```

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

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

Reinstall and more docs

- Now it is a good time to reinstall our R package and check again that everything works as expected.

- You should see following

```
library(statsAndBooze)
list_dates <- list(
  andrea = c("2023-04-01", "2023-04-02"),
  federico = c("2023-04-02", "2023-04-03"),
  chiara = "2023-04-02"
)
parsed_dates <- parse_dates(list_dates)
decide_happy_hour(parsed_dates)
[1] 2023-04-02
```

- We can also finish the docs by filling in some examples and complete the DESCRIPTION. Then CHECK and commit!

Unit testing

- The example reported in the previous slide informally shows that our R package works in a particular case.
- Now we want to formalise our expectations into **unit tests**!
- Why do we need unit testing? Two main reasons:
    1. We might want to test what happens with wrong inputs or other edge cases that can remain hidden for the end users;
    2. We want to ensure that all aspects of our package keep working even after refactoring its main functionalities.

Unit testing (cont)

- After loading `devtools`, you can run `use_testthat()` to setup the unit testing environment.

```
> use_testthat()
v Setting active project to 'D:/git/statsAndBooze'
v Adding 'testthat' to Suggests field in DESCRIPTION
v Setting Config/testthat/edition field in DESCRIPTION to '3'
v Creating 'tests/testthat/'
v Writing 'tests/testthat.R'
* Call `use_test()` to initialize a basic test file and open it
for editing.
```

- Then, we can run `use_test(<file>)` to create a new test file. So, for example, I can run `use_test("parse")`.

```
> use_test("parse")
v Writing 'tests/testthat/test-parse.R'
* Modify 'tests/testthat/test-parse.R'
```

Unit testing (cont)

- Now we need to edit the newly created file and write our unit test(s). First, we need to provide a short sentence that summarises the objective of the test. For example

```
test_that("parse_dates(): basic functionalities work", {
        expect_equal(2 * 2, 4)
})
```

- The tests are actually run using the R package `testthat` which exports several helper functions (`expect_length()`, `expect_message()`, `expect_error()`, ...) to test different aspects of our package (equality, differences, ...).
- Then, we need to write the corpus of the test comparing observed output and our expectation.

Unit testing (cont)

- For example:

```
test_that("parse_dates(): basic functionalities work", {
  input_strings <- list(
    andrea = "2023-04-03",
    marco = "2023-04-03"
  )
  expected_dates <- list(
    andrea = lubridate::as_date("2023-04-03"),
    marco = lubridate::as_date("2023-04-03")
  )
  expect_equal(parse_dates(input_strings), expected_dates)
})
```

- After loading the package (`load_all()`), we can run the new test interactively as any other R function.

Unit testing (cont)

- The same procedure can be repeated for the other function

```
> use_test("decide")
v Setting active project to 'D:/git/statsAndBooze'
v Writing 'tests/testthat/test-decide.R'
* Modify 'tests/testthat/test-decide.R'
```

- The actual test might look like

```
test_that("decide_happy_hour(): basic functionalities work", {
  beer_dates <- list(
    andrea = lubridate::as_date("2023-04-03"),
    federico = lubridate::as_date("2023-04-03"),
    chiara = lubridate::as_date("2023-04-03")
  )
  expect_equal(
    decide_happy_hour(beer_dates),
    lubridate::as_date("2023-04-03")
  )
})
```

Unit testing (cont)

- You should also test some pathological cases which might not be directly exposed to the regular end users

```
test_that("decide_happy_hour(): empty intersection", {
  beer_dates <- list(
    andrea = lubridate::as_date("2023-04-03"),
    marco = lubridate::as_date("2023-04-04")
  )
  expect_equal(
    decide_happy_hour(beer_dates),
    lubridate::as_date(numeric(0))
  )
})
```

- Similarly, you could develop a test to control the behaviour of the function in case of misspecified inputs.
- For example: what is the expected output when one or more of the input Date(s) is NA?

Unit testing (cont)

- The function `test()` (which is also defined in `devtools`) automatically runs all tests in a package and returns an informative output (with a funny comment on the results)

```
> test()
i Testing statsAndBooze
v | F W S  OK | Context
v |         2 | decide
v |         1 | parse

== Results ===============================
Duration: 0.5 s

[ FAIL 0 | WARN 0 | SKIP 0 | PASS 3 ]

You are a coding rockstar!
```

- The same behaviour occurs when you run R CMD check. Let's try and if everything looks right, we can also commit.

Github!

- The `use_github()` function can be used to automatically setup a Github project starting from our package.
- It runs the following steps (and many more, see the help page):
  1. Checks the initial state of the repo;
  2. Creates an associated repo on Github;
  3. Configures the appropriate remote so you can automatically run `push`/`pull` commands.
- **You need to be authenticated via a working PAT**.
- Let's test it together so we can check the messages!

Synch with a remote

- The `git pull`[2] command (or the blue arrow pointing downward in Rstudio) can be used to automatically synchronise your local work starting from a remote (the `origin` remote, by default).
- Now, if you click that arrow in Rstudio, the software should open a new panel with the following text

```
>>> C:/Program Files/Git/bin/git.exe pull
Already up to date.
```

- If you modify some files and you want to share your changes with others, you have to push them upstream. Let's try it together using the Rstudio GUI.

[2] `git pull` is a combination of `git fetch` (download data) and `git merge` (merge data). See here for more details.

Synch with a remote (cont)

- But what happens if the upstream has some changes you don't have in your local version? Your push will be rejected!
- For example, let's modify the DESCRIPTION file on Github and commit the change. Then, let's add one toy example to `parse_dates()` and run `document()`.
- If you commit and push, you should see the following

```
>>> git.exe push origin HEAD:refs/heads/main
To https://github.com/agila5/statsAndBooze.git
! [rejected]    HEAD -> main (fetch first)
error: failed to push some refs to [truncated...]
hint: Updates were rejected because the remote contains work
that you do not have locally. This is usually caused by another
repository pushing to the same ref. You may want to first
integrate the remote changes (e.g., 'git pull ...') before
pushing again. See the 'Note about fast-forwards' in 'git push
--help' for details.
```

Synch with a remote (cont)

- In the best case scenario, Git can harmonise the two versions!
- Therefore, following the hint, we can `pull` from upstream

```
>>> C:/Program Files/Git/bin/git.exe pull
From https://github.com/agila5/statsAndBooze
de846da..a79986f  main       -> origin/main
Merge made by the 'ort' strategy.
DESCRIPTION | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

(check the message!) and then you repeat the push

```
>>> git.exe push origin HEAD:refs/heads/main
To https://github.com/agila5/statsAndBooze.git
a79986f..2086e99  HEAD -> main
```

Synch with a remote (cont)

- But what can you do when your commit and the upstream are in two states that Git cannot automatically merge?
- For example, what happens if we replace the Description in DESCRIPTION both on Github and in our locale?
- If we commit and push the changes, we get the same rejection message as before.
- However, if we pull from our remote we see the following:

```
>>> C:/Program Files/Git/bin/git.exe pull
From https://github.com/agila5/statsAndBooze
2086e99..39b642e  main       -> origin/main
Auto-merging DESCRIPTION
CONFLICT (content): Merge conflict in DESCRIPTION
Automatic merge failed; fix conflicts and then commit the result
```

Synch with a remote (cont)

- The DESCRIPTION file should look like

```
Package: statsAndBooze
Title: Find the best day to have a beer!
Version: 0.0.0.9000
Authors@R: [omitted...]
<<<<<<< HEAD
Description: DEF
=======
Description: ABC
>>>>>>> 39b642e405fd0bdd0efef9fb79b182dab278e66d
```

- As we can see, Git adds conflict-resolution markers.
- The upstream version of the file (i.e. what is saved on Github) is summarised between <<<<< and ======, while your local version lies in the bottom part.
- Using any text editor you can manually fix the conflict. Then add the files and repeat the commit plus push process.

Synch with a remote (cont)

- If the automatic process with the GUI in Rstudio does not work, you need to complete the procedure using the shell.
- More precisely, you need to `git add` all files that presented a conflict. Then, you have to create a `commit` and finally you can push to the remote (i.e. Github).

I want to work with my friends!

- If you want to add collaborators to your (public or private) Github repo, you can execute the following steps:
  1. Go to the Github page of your repo and click Settings;
  2. Then, click on Collaborators in the section named Access;
  3. Finally, click on the green button named Add people and write the name or the ID of the person you want to invite.

- Let's try it together! Each group should nominate a team leader that must add all the other team members (plus me!) as collaborators to his/her repository.

- Please note that you can customise the permissions of your collaborators. More details here.

I want to work with my friends!

- Clearly, when several people work on the same Github repo, it's really easy to create a merge conflict. Let's try!
- Each group can simulate the process:
  - First, one of you should modify a file and commit the changes to Github. The process should work out smoothly.
  - Then, another one modifies the same part of the same file and commit the changes. If you push now, Github should reject your changes.
  - If the second guy pulls the new changes, he/she will get a merge conflict. Fix it together and push the new version of that file! Finally, repeat the process changing the roles.

Git branching (😢)

- Following the Git Book, we can say that *“Branching means you diverge from the main line of development and continue to do work without messing with that main line”*.
- This is a quite convenient feature, especially when several people work on the same project.
- Unfortunately, we cannot cover this topic here, but I refer you to the manual for more details.
- If you want, we can also organise a quick (1h) extra class after the Easter holidays.

THE END!

E quindi uscimmo a riveder le stelle

Dante Alighieri, Inferno XXXIV, 139