UCLouvain

SMCS

Write and publish your work in an R package
Dec. 2024

Table of contents I

Table of contents II

UCLouvain

This training is organised by the SMCS, Statistical Methodology and Computing Service, UCLouvain's technological platform.

Its content is subject to the UCLouvain regulations on the valorisation of intellectual property and the valorisation of works covered by copyright legislation and produced within UCLouvain. As such, the distribution of the attached materials is not allowed without our permission.

The first version of this training course was written by J. Chau and M. Martin in 2018. Its content has been adapted over the years by our consultants.

Services provided by the SMCS

Consulting Involvement at any point in the life of a project involving data analysis, from its conception – the perfect time to contact us and allow us to offer quality methodological and statistical support - to the interpretation and presentation of results.

Training Training courses in multiple statistical methods and tools, which are sometimes organised mutliple times a year and last from half a day to a week or even several months, some of which may lead to a university certificate. Possibility of tailor-made training courses.

Surveys Help in conducting questionnaire-based surveys, provided through training courses, personalised advice or by taking over management of the entire project.

IT tools (for UCLouvain members only) Provision of a wide range of IT tools and documentation.

Get started

Why you should write an R package?

PROS

- **Good practices and automation are key for time-saving and reproducibility**
  - Avoid coding errors and multiple versions of your work
  - Functions, data and package documentation altogether
  - Conventions and tools standardisation
  - Available **tests** and **checks** of the package
  - keep track of changes with **code versionning**

- **Portable code**
  - Easier to send your codes on a remote server
  - Easier to share your codes within your team or carry out a group work (with GitHub)
  - Open your code to the R community (extra testing for bugs, meet new needs, etc.)
  - **Publish** and value your coding work (along with your articles)
  - Deploy Shiny apps and a website of your package online

Why you should write an R package?

CONS

- More upstream work
- Must pass checks/tests and meet the standards
- Maintainance cost
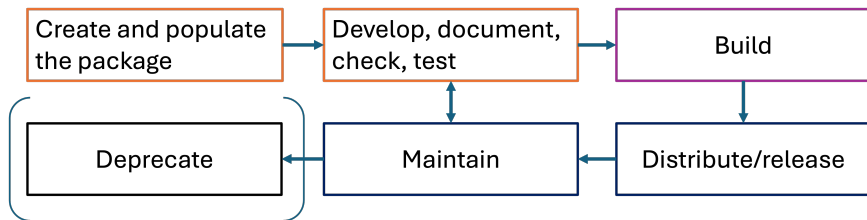
Work with existing R packages and terminology

- Different **repositories**: CRAN (21749), Bioconductor (2289), R-forge , Github, ...
- Example: lubridate
- **Install**: downloads the packages from the repositories and installs them to the 1st directory in .libPaths() by default

```
# Installation from popular repositories
utils::install.packages("tidyverse") # CRAN
BiocManager::install("SummarizedExperiment") # Bioconductor
remotes::install_github("david-barnett/microViz") # GitHub
```

- **Attach**: loads the *namespace* of the package with name package and attaches it on the search list

```
# load and attach the package
base::library(tidyverse)
```

Lifespan of an R package


```
Create and populate  →  Develop, document,  →  Build
the package             check, test
                             ↕                    ↓
Deprecate  ←  Maintain  ←  Distribute/release
```

Different package states

in-memory: after library(). The package is loaded in your R session.

installed: after install.packages(). The package is installed on your machine.

bundled/binary: intermediary state which gives you a single file (e.g. .tar.gz) which you can use to install your package on another machine.

source: The directory we are working on during package development.

Demo package used for the training

Let's build a package !

- During this training, we will create from scratch a demonstration R-package called `moviesdemo`
- We will use a dataset containing metadata (e.g. title, popularity...) on 3077 movies from The Movie Database (TMDb)
- The goal of the package is to advise similar movies to watch based on a movie selected by the user or on keywords

Useful resources/references

- *'Writing R Extensions'*, the official guide (https://rstudio.github.io/r-manuals/r-exts/)
- *'R Packages (2e)'* by Hadley Wickham and Jennifer Bryan (https://r-pkgs.org/)
- *'Package Development::CHEATSHEET'* (https://rstudio.github.io/cheatsheets/package-development.pdf)
- *'Bioconductor Packages: Development, Maintenance, and Peer Review'* (https://contributions.bioconductor.org/)
- *'Advanced R'* (2nd edition) by Hadley Wickham (https://adv-r.hadley.nz/index.html)
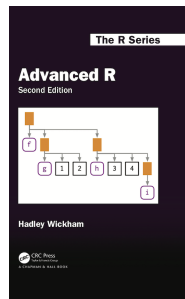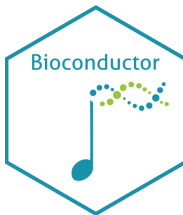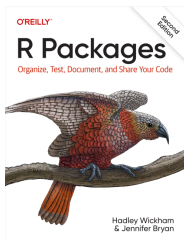
Package creation and structure

# Overview of package creation and development

With `devtools` functions

Steps:

1. Prepare R codes, data and their documentation
2. Create the package structure and populate it with files, document and test
   - `create()`: creates a new package skeleton
   - `load_all()`: loads a package in memory (without installing it)
   - `document()`: documents the package with `roxygen`
   - `test()`: executes `testthat` tests
3. Check for errors
   - `check()`: checks and builds a source package
4. Build and install
   - `build()` and `install()`

Package creation from RStudio

Using the package creation workflow: `File > New Project > New Directory > R package`
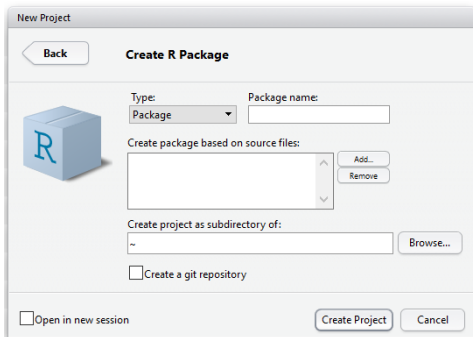
- Set the package name and directory
- Select source files (= R scripts)
- Create a git repository (optional)

Package name

- The choice of the package name is important for your package visibility
- Name it with letters, number and period only
- It must start with a letter and cannot end with a period

General Package Structure

Figure 2: source:
https://rstudio.github.io/cheatsheets/package-development.pdf

Initial structure of `moviesdemo`
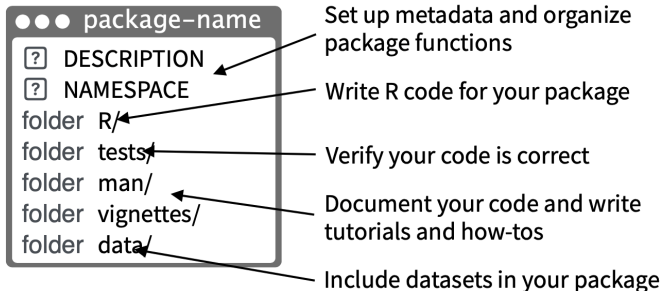
... Just after its creation

```
moviesdemo_init
+-- DESCRIPTION
+-- NAMESPACE
+-- R
|   \-- hello.R
+-- man
|   \-- hello.Rd
\-- moviesdemo.Rproj
```

Exercices - getting started (I)

- for Windows users, install **Rtools**
  (https://cran.r-project.org/bin/windows/Rtools/)
- Install and attach devtools and other useful packages

```
install.packages(c("devtools","roxygen2",
                  "testthat","knitr",
                  "usethis", "rmarkdown",
                  "purrr"))
```
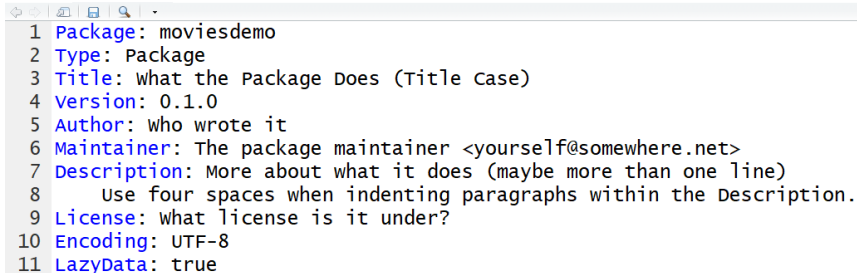
- Navigate through the files (R scripts, data, doc) from the
  start_moviesdemo folder
- Create the initial structure of the package moviesdemo with
  RStudio
- Populate the package with the R scripts (5 files) in the R folder
  and remove hello.R

Exercices - getting started (II)

- Create a `data` directory and store `movies_DB.rda` in it
- Inspect the `movies_DB` dataset
- Run `devtools::load_all()` within the package project
- Try to run the functions `advise_movie()` and `advise_movie_k()` with appropriate arguments. e.g.:

```
advise_movie(similar_to = "Interstellar", how_many = 2)
advise_movie_k(c("princess", "magic"), how_many = 3)
```

- Open `R/advise_movie.R` and compile the function (`Source`); then run `devtools::load_all()`. Try to interpret the warning message and remedy to it.

DESCRIPTION file

Mandatory DCF file that stores the package metadata. Specifies dependencies, who can use it and how (license), whom to contact in case of problems, . . .

```
 1  Package: moviesdemo
 2  Type: Package
 3  Title: What the Package Does (Title Case)
 4  Version: 0.1.0
 5  Author: Who wrote it
 6  Maintainer: The package maintainer <yourself@somewhere.net>
 7  Description: More about what it does (maybe more than one line)
 8      Use four spaces when indenting paragraphs within the Description.
 9  License: what license is it under?
10  Encoding: UTF-8
11  LazyData: true
```

DESCRIPTION file

- `Package` (*): Package name
- `Title` (*): One line description of the package
- `Description`: Multiple sentences short description of the package
- `Author`[1] (*): Package authors
- `Maintainer`[1] (*): Package maintainer
- `Version` (*):
  - Released version: {'<major>.<minor>.<patch>'}
  - In-development package: add a 4th component, starts at 0.0.0.9000
- `License`: Important for the package release. Explain who can and how to use the package (e.g. GPL-3). A LICENSE file can be added for more information
- `LazyData`: If `TRUE` (default), the datasets are lazily loaded

---

[1]or field 'Authors@R': Package Authors ("aut"), Creator and package *Maintainer* ("cre"), Contributors ("ctb"), ... Comprehensive code list:
http://www.loc.gov/marc/relators/relaterm.html

DESCRIPTION file - Dependencies

Possible fields

Imports Packages with imported objects that are *loaded* but which do not need to be *attached* (only available to the package that imported them).

Depends Mostly used to refer to the compatible R version(s) and packages that should be *attached* when loading yours. If attached, all of its its dependencies are available to the users.

Suggests Packages that are not required for installation but can be necessary (e.g. for datasets, to run tests or build the vignette, ...).

DESCRIPTION file - Dependencies

- Versioning to specify a minimum package version: e.g.:
  `knitr(>=1.17)`
- Illustration of `Suggests`:

Case of a specific function needing a package from `Suggests`: test if the
package is installed in the function R script:

```
1   FUN <- function(x) {
2     if (!requireNamespace("suggestedPackage", quietly = TRUE)) {
3       stop("suggestedPackage installation is necessary for function FUN") }
4   }
```

In `moviesdemo/DESCRIPTION`:

- Change the `Title` and `Description` fields
- Replace the `Author` and `Maintainer` fields by

```
Authors@R:
    person("First", "Last",
           "first.last@example.com",
           role = c("aut", "cre"))
```

- Fill in the license with the function

```
usethis::use_gpl_license(version = 3, include_future = TRUE)
```

NAMESPACE file

- Used to load or attach objects from other packages and export your functions
- Automatically updated when documenting with roxygen2 (NAMESPACE should not be edited by hand)

Illustration:

```
# Exports fun1 and fun2 from your package
export(your_fun1, your_fun2)
# Imports all objects from package foo
import(foo)
# Imports objects `foo_fun1` and `foo_fun2` from `foo`
importFrom(foo, foo_fun1, foo_fun2)
```

Rem: only exported objects need to be documented

The .Rbuildignore file

- Motivation: Files from the source package appearing in
  .Rbuildignore are not included in the bundled or binary package
- devtools::use_build_ignore("files"): generates the
  regular expression in the .Rbuildignore file

Illustration (default):

```
^.*\.Rproj$
^\.Rproj\.user$
^LICENSE\.md$
```

Package subdirectories

Main subdirectories

R R source code (exported and internal functions) and roxygen2 documentation

man Manuals for the objects (functions, data, classes, package, ...)

data Data files. Allowed formats: R, tables (.tab, .txt, .csv) or R objects (.RData or .rda)

tests Package unit tests with `testthat`

inst Contains `CITATION` file (`?citation`), `doc/` (additionnal documents), `extdata/` (raw data), ...

vignettes Package vignettes

src Compiled code

...

R functions

- All R code goes in the folder /R
- File names should be meaningful
- One file can contain multiple functions
- File names should end with .R or .r

Code style

- Automatically restyles the entire package with
  `styler::style_pkg()`

- Warning about potential style, syntax or semantic problem:
  `lintr::lint_package()`

- Conventions concerning object names, spacing, { }, comments,
  indentation, etc.: http://adv-r.had.co.nz/Style.html and
  https://style.tidyverse.org/

R functions - Good practices

Top level code rules in the scripts:

- Never use `library()` or `require()` since packages will not be loaded and it modifies the search path
- Never `source()` your code since it modifies the current environment. Instead use `devtools::load_all()` during the package development.
- Use carefully the global `options()`, the graphical parameters `par()`, all functions modifying default directories (e.g. `libPaths()` or `setwd()`), you must reset after use with `on.exit()`
- use the syntax `package::object` to explicit the package to be used
- Use `devtools::load_all()` during the package development to avoid re-installations after R code modifications

R functions - Input arguments check

errors and *warnings* should be meaningful

```
average_fun <- function(numbers, type = c("mean","median")){
  switch(type,
         mean = mean(numbers),
         median = median(numbers))
}

average_fun(numbers = c(1, 4, 6, 5, 5, 2, 1),type="mean")

[1] 3.428571

average_fun(numbers = c(1, 4, 6, "a", 5, 2, 1),type="mean")

[1] NA

average_fun(numbers = c(1, 4, 6, 5, 5, 2, 1),type="meen")
average_fun(numbers = c(1, 4, 6, 5, 5, 2, 1))

Error in switch(type, mean = mean(numbers), median = median(numbers)): EXPR must
```

R functions - Input arguments check

```r
average_fun <- function(numbers, type = c("mean","median")){
  ## check for numbers
  if (!is.numeric(numbers)){
    stop(deparse(substitute(numbers)), " is not numeric")
  }
  ## check for type
  type <- match.arg(type)

  switch(type,
         mean = mean(numbers),
         median = median(numbers))
}

average_fun(numbers = c(1, 4, 6, "a", 5, 2, 1),type="mean")

Error in average_fun(numbers = c(1, 4, 6, "a", 5, 2, 1), type = "mean"): c(1, 4,

average_fun(numbers = c(1, 4, 6, 5, 5, 2, 1),type="meen")

Error in match.arg(type): 'arg' should be one of "mean", "median"

average_fun(numbers = c(1, 4, 6, 5, 5, 2, 1))

[1] 3.428571
```

Data

Data

Data can be included to provide functions' examples but some packages are only created to distribute data = **data packages**

Including data in the package:

Exported data Available to the user, stored in data/. Allowed formats: R, tables (.tab, .txt, .csv) or R objects (.RData - recommended - or .rda) (see ?data)

Internal data Not available to the users, stored in R/sysdata.rda

Raw data Available to the users, stored in inst/extdata

Data

- Tips:
  - Lazydata:TRUE is automatically set in the DESCRIPTION file: datasets do not occupy memory until used
  - The source package (after build) should occupy less than 5 MB on disk (except for data packages) –> compress = "bzip2" or "xz" in save() or usethis::use_data() reduce the size of the .rda or .RData files

- Create package data (illustration from a package repository):

```r
x <- sample(1000)
usethis::use_data(x, compress = "xz") # data in data/x.rda
usethis::use_data(x, internal = TRUE, compress = "xz") # data
↪  in R/sysdata.rda
```

Documentation

Example of function documentation

We want to produce this for the function `advise_movie()`:

advise_movie {moviesdemo}   R Documentation

Advise movies based on another movie

Description

'advise_movie' takes as input a movie from the movie database and gives as output a number of movies that are similar.

Usage

```
advise_movie(similar_to, how_many, draw_scores = FALSE, ...)
```

Arguments

`similar_to`   character, movie title from the database.

`how_many`   integer, how many movies to advise.

`draw_scores` if 'TRUE', draws a barplot with the similarity scores.

`...`   additional arguments (for now, only 'weights').

Value

A 'list' with the following elements: 'movie_title': advised movie title(s), 'plot': plot(s) of advised movie title(s), 'movie_ids': line number in the 'movies_DB' database of the advised movie(s), 'scores': similarity scores of the advised movie(s)

Examples

<u>Run examples</u>

```
suggestions <- advise_movie(similar_to = "Interstellar", how_many = 3,
draw_scores = TRUE, weights = c("genre"=1, "popularity"=1, "rating"=1,
```

Documentation - overview

- Useful both for package developers and users

Different ways to document your package:

Exported R objects exported R objects (functions, data, ...) have to be documented with roxygen2; help accessed with `help()`

Reference manual Compilation of the package `DESCRIPTION` and help pages

Vignettes Case studies, long-form guides to your package (in `vignettes/` directory, RMarkdown or Quarto format)

Markdown files `README.md` (Why should I use it? How do I use it? How do I get it?) and `NEWS.md` (changes over time)

Package website with GitHub and the `pkgdown` package

R documentation (`.Rd`) format

R documentation files format

- Exported R objects have to be documented in `.Rd` (R documentation) files format.
- `Rd` is a simple markup language, loosely based on LaTeX
- Output formats: HTML, LaTeX, pdf, ...
- All manuals go in the folder `/man`, one file per exported object

`roxygen2` **comments**

- In-Line documentation with `roxygen2` to generate the Rd files: roxygen2 comments (starting with #') at the beginning of the `.R` scripts
- More info on `roxygen2`: http://cran.r-project.org/web/packages/roxygen2/vignettes/roxygen2.html.

In `moviesdemo`, only `advise_movie` and `advise_movie_k` are documented, `sim_genres` and `sim_producers` are considered as **internal functions** and are not documented/exported.

advise_movie.Rd file

```
% Generated by roxygen2: do not edit by hand
% Please edit documentation in R/advise_movie.R
\name{advise_movie}
\alias{advise_movie}
\title{Advise movies based on another movie}
\usage{
advise_movie(similar_to, how_many, draw_scores = FALSE, ...)
}
\arguments{
\item{similar_to}{character, movie title from the database.}
\item{how_many}{integer, how many movies to advise.}
\item{draw_scores}{if `TRUE`, draws a barplot with the similarity scores.}
\item{...}{additional arguments (for now, only `weights`).}
}
\value{
A `list` with the following elements: `movie_title`: advised movie title(s),
`plot`: plot(s) of advised movie title(s),
`movie_ids`: line number in the `movies_DB` database of the advised movie(s),
`scores`: similarity scores of the advised movie(s)
}
\description{
`advise_movie` takes as input a movie from the movie database and gives as output a number
of movies that are similar.
}
\examples{
suggestions <- advise_movie(similar_to = "Interstellar", how_many = 3,
draw_scores = TRUE, weights = c("genre"=1, "popularity"=1, "rating"=1,
"production company"=1))
}
```

```r
#' Advise movies based on another movie
#'
#' `advise_movie` takes as input a movie from the movie database and gives as output a number
#' of movies that are similar.
#'
#' @param similar_to character, movie title from the database.
#' @param how_many integer, how many movies to advise.
#' @param draw_scores if `TRUE`, draws a barplot with the similarity scores.
#' @param ... additional arguments (for now, only `weights`).
#'
#' @returns
#' A `list` with the following elements: `movie_title`: advised movie title(s),
#' `plot`: plot(s) of advised movie title(s),
#' `movie_ids`: line number in the `movies_DB` database of the advised movie(s),
#' `scores`: similarity scores of the advised movie(s)
#'
#' @examples
#' suggestions <- advise_movie(similar_to = "Interstellar", how_many = 3,
#' draw_scores = TRUE, weights = c("genre"=1, "popularity"=1, "rating"=1,
#' "production company"=1))
#'
#' @import graphics
#' @importFrom purrr map_dbl
#'
#' @export
advise_movie <- function(similar_to, how_many, draw_scores = FALSE, ...){

  dots <- list(...)
  ...
```

Elements of Roxygen2 comments

- Introduction block (Mandatory) is text without tag
  - first sentence: **title**
  - 2nd paragraph: **description**
- Other blocks with tags (main ones):
  - `@param` name description: describes function parameters
  - `@examples`: executable R script applying the function. You may wish to use `\dontrun{}` to avoid errors check (e.g. if you plan to show a code which actually would produce an error).
  - `@returns`: describes the outputs of the function
  - `@export` and `@import` (or `@importFrom`): specifies NAMESPACE imports/exports

Data documentation

- Objects in `data/` are always exported and need documentation
- Documenting data is similar to documenting a function
- Procedure:
    - Create a `.R` file and save it in `R/`, e.g. `R/movies_demo_DB.R`
    - Document the `.R` file similarly than for R functions.
    - See http://r-pkgs.had.co.nz/data.html for more details

Example for dataset `movies_DB`: R/movies_DB.R

```
#' TMDB Movies Dataset 2024
#'
#' data are from a filtered version of the Full TMDB Movies Dataset 2024 (1M
↪  Movies)
#'
#' The variables are as follows:
#'    \describe{
#'      \item{title}{character, title of the film.}
#'      \item{genres}{string of characters, genres of the film.}
#'      \item{popularity}{numeric, popularity of the film in terms of
↪  views.}
#'      \item{vote}{numeric, voted rating of the film between 0 and 10.}
#'      \item{language}{factor, original language.}
#'      \item{producers}{string of characters, production companies.}
#'      \item{release_date}{date, release date of the film.}
#'      \item{runtime}{numeric, runtime in minutes.}
#'      \item{plot}{character, plot summary of the film.}
#'      \item{tagline}{character, tagline of the film.}
#'      \item{keywords}{character, keywords of the film.}
#'    }
#' @format A data frame with 3077 rows and 11 variables.
#' @source
↪  \url{https://www.kaggle.com/datasets/asaniczka/tmdb-movies-dataset-2023-930k-↵
"movies_DB"
```

Example for dataset `movies_DB`: `man/movies_DB.Rd`

```
\docType{data}
\name{movies_DB}
\alias{movies_DB}
\title{TMDB Movies Dataset 2024}
\format{
A data frame with 3077 rows and 11 variables.
}
\source{
\url{https://www.kaggle.com/datasets/asaniczka/tmdb-movies-dataset-2023-930k-movies}
}
\usage{
movies_DB
}
\description{
data are from a filtered version of the Full TMDB Movies Dataset 2024 (1M Movies)
}
\details{
The variables are as follows:
    \describe{
      \item{title}{character, title of the film.}
      \item{genres}{string of characters, genres of the film.}
      \item{popularity}{numeric, popularity of the film in terms of views.}
      \item{vote}{numeric, voted rating of the film between 0 and 10.}
      \item{language}{factor, original language.}
      \item{producers}{string of characters, production companies.}
      \item{release_date}{date, release date of the film.}
      \item{runtime}{numeric, runtime in minutes.}
      \item{plot}{character, plot summary of the film.}
      \item{tagline}{character, tagline of the film.}
      \item{keywords}{character, keywords of the film.}
  }
}
\keyword{datasets}
```

Exercices - part I

1. Start by inspecting then deleting the NAMESPACE file

2. **Documenting package elements**

- After each step 3 to 5, inspect the roxygen2 comments and run:
  - devtools::document() to generate the doc, and
  - pkgload::dev_help() to inspect the generated .Rd file
    (e.g. dev_help(advise_movie)).

3. **Documenting functions**

- Document advise_movie()
  - Copy-paste text in doc/advise_movie.txt at the top of
    R/advise_movie.R
- Document advise_movie_k()
  - Copy-paste text in doc/advise_movie_k.txt at the top of
    R/advise_movie_k.R
  - Complete the documentation

4. **Documenting data**

- Copy-paste text in doc/data.txt into R/movies_DB.R

5. **Documenting the package**

- run usethis::use_package_doc() then use information in doc/moviesdemo.txt to complete the doc

6. Further steps:

- Inspect the generated NAMESPACE file
- Run devtools::check() and try to solve errors/warnings[1]

---

[1]The 'purrr' and 'methods' packages need to be specified in either 'Imports' or 'Depends' fields in the 'DESCRIPTION' file.

Load/attach other packages

- Motivations:
  - `Imports` in the `DESCRIPTION` file loads imported objects but does not attach them (contrarily to `Depends`)
  - `library()` or `require()` not allowed in .R files
- load/attach a package to be used in your package functions
  - Use the following syntax: `package::function`
  - Use `@import` in roxygen2 comments to automatically update the `NAMESPACE` file with `devtools::document()`

Export your functions

Use `@export` to write in the `NAMESPACE` file which of your functions should be available outside your package without conflict.

Illustration: `sessionInfo()`

```
sessionInfo()

R version 4.4.2 (2024-10-31)
Platform: x86_64-apple-darwin20
Running under: macOS Sonoma 14.6.1

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.4-x86_64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.4-x86_64/Resources/lib/libRlapack.dylib;  LAPACK version

locale:
[1] en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: Europe/Brussels
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] devtools_2.4.5 usethis_3.1.0 fs_1.6.5       knitr_1.49

loaded via a namespace (and not attached):
 [1] miniUI_0.1.1.1    jsonlite_1.8.9    crayon_1.5.3      compiler_4.4.2
 [5] promises_1.3.2    Rcpp_1.0.13-1     later_1.4.1       yaml_2.3.10
 [9] fastmap_1.2.0     mime_0.12         R6_2.5.1          htmlwidgets_1.6.4
[13] profvis_0.4.0     shiny_1.9.1       rlang_1.1.4       cachem_1.1.0
[17] httpuv_1.6.15     xfun_0.49         pkgload_1.4.0     memoise_2.0.1
[21] cli_3.6.3         magrittr_2.0.3    digest_0.6.37     rstudioapi_0.17.1
[25] xtable_1.8-4      remotes_2.5.0     lifecycle_1.0.4   vctrs_0.6.5
[29] evaluate_1.0.1    glue_1.8.0        urlchecker_1.0.1  sessioninfo_1.2.2
[33] pkgbuild_1.4.5    rmarkdown_2.29    purrr_1.0.2       tools_4.4.2
[37] ellipsis_0.3.2    htmltools_0.5.8.1
```

Package documentation with vignette(s)

Usually the most exhaustive and informative form of documentation

- Motivation
  - Long-form tutorials usually written in RMarkdown or Quarto
  - A vignette can e.g. describe how to combine different functions in the package to solve a complex problem
- Setup
  - Install packages `rmarkdown` and `knitr`
  - Install pandoc -> http://pandoc.org/installing.html
  - Run `usethis::use_vignette("my_vignette")` creating necessary files/folders
  - Vignettes are compiled at build time
- Markdown and knitr
  - Simple text formatting language combining text and R code
  - See http://r-pkgs.had.co.nz/vignettes.html for more info
- Illustration: lubridate vignette

Compiled Code

- Speed up your code by including C or C++ code in your package with Rcpp (and RcppArmadillo or RcppEigen) or cpp11 for packages using tidyverse.
- Source and header files for compiled code go to the src/ directory
- For more details, see https://adv-r.hadley.nz/rcpp.html

Build, install and load

Exercices

Once checked with `devtools::check()` and no more error/warning:

- Build and install the package
- Load the package and explore its components in RStudio: help pages, function examples, …
- Generate the reference manual with `devtools::build_manual()`
- Modify other entry arguments of `build()` and `install()` and see how it impacts the build/installation

```
# build the package
devtools::build()
# install the package
devtools::install()
# Create package pdf manual
devtools::build_manual()
# load the package
library(moviesdemo)
```

Package release

Package first release

Main steps:

- `devtools::check()` with no error/warning
- Add Git/GitHub (mandatory for CRAN and Bioconductor)
- Determine the release type to modify the version number with `usethis::use_version()`
- Add `README.md` with installation instructions
- Add `NEWS.md` (`usethis::use_news_md()`) to describe how the package changes
- Double check `DESCRIPTION` file (license, authors, title, description, etc.)

Specific to CRAN

- Generate a checklist as a GitHub issue:
  `usethis::use_release_issue()` (can change over time)
- Create `cran-comments.md`, to communicate with CRAN during
  package submission: `usethis::use_cran_comments()`
- Look at extra checks:
  https://github.com/DavisVaughan/extrachecks
- Run automated and manual tests, then post package to CRAN:
  `devtools::release()`

For more infos: https://r-pkgs.org/release.html#sec-release-initial

Specific to Bioconductor

- Repository dedicated to **bioinformatics**
- Different types of packages: *Software*, *Experiment Data*, *Annotation* and *Workflow*.
- Extra review process for package content
- biocViews field (keywords) in DESCRIPTION
- Packages must also pass BiocCheck::BiocCheckGitClone() and BiocCheck::BiocCheck('new-package'=TRUE)
- Illustration of packages under review: https://github.com/Bioconductor/Contributions/issues

For more info: https://contributions.bioconductor.org/

Advanced tools

Formal testing with `testthat`

- Prerequisite: install the `testthat` package
- Motivation
  - Effective and automatic testing to see if code is (still) working properly
  - Investment for future developments: confidently make changes without breaking other functionalities, fewer bugs, etc.
- Test structure
  - `testthat` allows you to test a piece of code via the concept of **expectation** with functions starting with `expect_`
  - An expectation takes two arguments :
    1) what you actually have (*reality*)
    2) what you expect to have (*expectation*)
  - Test is considered passed if *reality* meets *expectation*.

Basic examples of expectations (1)

```r
library(testthat)

# expect_equal
expect_equal(10,10) ## OK
expect_equal(10, 10 + 1E-7) ## OK
expect_equal(10, 11) ## ERROR


Error: 10 not equal to 11.
1/1 mismatches
[1] 10 - 11 == -1

# Match for character vectors
string <- "Testing is fun"
expect_match(string, "Testing is fun")
expect_match(string, "testing is fun") # case sensitive


Error: `string` does not match "testing is fun".
Actual value: "Testing is fun"
```

Basic examples of expectations (2)

```
# Inspect printed output
expect_output(str(list(1:10, letters)), "List of 2") ## OK
expect_output(str(list(1:10, letters)), "List of 4") ## ERROR


Error: `str\(list\((1:10, letters\)\)` does not match "List of 4".
Actual value: "List of 2\\n \$ : int \[1:10\] 1 2 3 4 5 6 7 8 9 10\\n \$ : chr \[

expect_message(message("Hello"), "Hello")

# You may have to change the text to "Production de NaN"
# if your R version is in French
expect_warning(log(-1), "NaNs produced")

# You may have to change the text to "argument non numérique
# pour un opérateur binaire" if your R version is in French
expect_error(1 / "a", "non-numeric argument")
```

Example of tests for `moviesdemo`

A test created with `test_that()` groups multiple expectations to test a (simple) function

```
# tests for sim_genres
test_that("Output sim_genres measures", {
  expect_type(sim_genres(movies_DB$title[1], movies_DB$title[2]), "double")
  expect_equal(sim_genres(movies_DB$title[1], movies_DB$title[1]), 1)
  expect_error(sim_genres(NA, movies_DB$title[1]))
})
# tests for advise_movie
test_that("Output advise_movie function", {
  out <- advise_movie(movies_DB$title[1], 5, weights = rep(1,4))
  expect_output(str(out),"with 3 slots")
  expect_match(out@movie_title[1], movies_DB$title[1606])
  expect_equal(length(out@movie_title), 5)
  expect_type(out@movie_title, "character")
  expect_error(advise_movie(movies_DB$title[1], NA),
               "Argument 'how_many' should be a number...")
})
```

Exercices

- Initiate test files/folders with usethis::use_testthat():
  creates tests/testthat/ and tests/testthat.R
- From the tests_fun/test-functions.txt file, write your test
  functions in R files[1] starting with 'test-' and place them inside
  the tests/testthat/ folder:
    - test-sim_genres.R, test-sim_producers.R and
      test-advise_movie.R respect. for sim_genres, sim_producers
      and advise_movie functions
- Execute each line of code of expectations enclosed in
  test_that("...", { expectations }
- Run
  testthat::test_file("tests/testthat/test-sim_genres.R")
  to run tests for sim_genres
- Run tests at the package level with devtools::test()

---

[1]Test files should match the organisation or R files in the 'R/' directory.

Object-Oriented Programming

- Multiple OOP systems (S3, S4, R6, ...)
- Concept of **encapsulation**: the user doesn't need to worry about details of an object because they are encapsulated behind a standard interface.
- Concept of **polymorphism**:

```
diamonds <- ggplot2::diamonds
class(diamonds$carat)

[1] "numeric"

summary(diamonds$carat)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.2000  0.4000  0.7000  0.7979  1.0400  5.0100

summary(diamonds$cut)

     Fair      Good Very Good   Premium     Ideal
     1610      4906     12082     13791     21551
```

Object-Oriented Vocabulary

class Type of an object. Class defines what an object is (e.g. `numeric` for `diamonds$carat`)

method Implementation for a specific class. Methods describe what that object can do (e.g. if object of class `numeric`, then compute the mean, median, ...)

generics A generic function defines an interface, which uses a different implementation depending on the class of an argument (e.g. `summary`)

Illustration: define the S4 class `advice`

```
# An S4 class to represent a movie advice
methods::setClass("advice", # define the S4 class
          slots = c( # describes the names and classes of the slots
            movie_title = "character",
            plot = "character",
            keywords = "character"
          ),
          prototype = list( # default values for each slot
            movie_title = NA_character_,
            plot = NA_character_,
            keywords = NA_character_
          )
)
```

Define a method for an existing S4 generic (show)

```r
methods::setMethod("show", "advice", function(object) {
  nn <- length(object@movie_title)
  titles <- paste(object@movie_title, collapse = ", ")
  plots <- paste(paste(object@movie_title,object@plot,
                       sep = ": "),collapse = "\n")
  cat(methods::is(object)[[1]], " with ",
      nn," suggestion(s)","\n",
      "=========================","\n",
      "selected movie movie_title(s): ", titles, "\n",
      "=========================","\n",
      "plots:  ",plots, "\n",
      sep = "")
})
```

Helper for class advice

```
# helper (user friendly) to create a new advice
advice <- function(movie_title,
                   plot = NA_character_,
                   keywords = NA_character_) {

  methods::new("advice", movie_title = movie_title,
               plot = plot,
               keywords = keywords)
}
new_adv <- advice("movie1")
new_adv


advice with 1 suggestion(s)
==========================
selected movie movie_title(s): movie1
==========================
plots:  movie1: NA

advice(32) # not working (internal validation)


Error in validObject(.Object): invalid class "advice" object: invalid object for
```

Setter and getter of the movie title for class advice

```
# getter
methods::setGeneric("movie_title", function(object) standardGeneric("movie_title"))
methods::setMethod("movie_title", "advice", function(object) object@movie_title)
# setter
methods::setGeneric("movie_title<-", function(object, value) standardGeneric("movie_title<-"))
methods::setMethod("movie_title<-", "advice", function(object, value) {
  object@movie_title <- value
  methods::validObject(object)
  object
})


movie_title(new_adv)


[1] "movie1"


movie_title(new_adv) <- "new_title"
new_adv


advice with 1 suggestion(s)
=========================
selected movie movie_title(s): new_title
=========================
plots:  new_title: NA
```

Software development principles

- Use of an **IDE** (Integrated Development Environment): RStudio for R package development
- Use **Version control** with Git and GitHub (or equivalent): strongly recommended
    - Git: version control system installed on your local computer
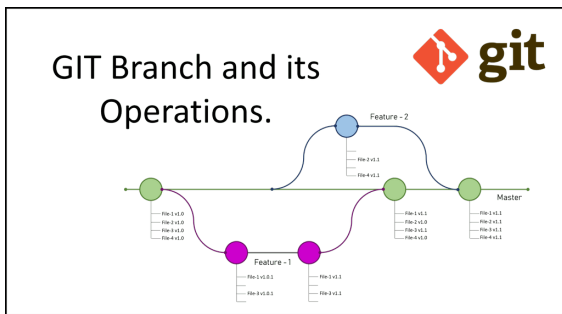   - GitHub: remote host

Figure 3: source:
https://ucdavisdatalab.github.io/adventures_in_data_science/version-control.html#git-branching

Version control and Continuous integration

Version control principles

- Manages the evolution of a set of files — called a *repository* — in a highly structured way
- Multiple versions can co-exist (branches) and be merged
- Enables multiple people to simultaneously work on a single project
- Access to historical versions of your project
- GitHub: bug reports and feature requests, collaboration, package distribution (`remotes::install_github()`), package website, continuous integration, ...
- More infos on Git/GitHub and its integration with R: https://happygitwithr.com/index.html

Continuous integration

Set up specific package development tasks to happen automatically when you push new work to your hosted repository (check and re-build the package) e.g. with GitHub Actions

Summary

Final structure of the `moviesdemo` package

```
moviesdemo
+-- DESCRIPTION
+-- LICENSE.md
+-- NAMESPACE
+-- R
|   +-- advice_class.R
|   +-- advise_movie.R
|   +-- advise_movie_k.R
|   +-- movies_DB.R
|   +-- moviesdemo.R
|   +-- sim_genres.R
|   \-- sim_producers.R
+-- data
|   \-- movies_DB.rda
+-- man
|   +-- advice.Rd
|   +-- advise_movie.Rd
|   +-- advise_movie_k.Rd
|   +-- movies_DB.Rd
|   \-- moviesdemo-package.Rd
+-- moviesdemo.Rproj
\-- tests
    +-- testthat
    |   +-- _snaps
    |   +-- test-advise_movie.R
    |   +-- test-sim_genres.R
    |   \-- test-sim_producers.R
    \-- testthat.R
```

Main steps for package creation/development

1. Prepare R codes, data and their documentation
2. Create the package structure and populate it with files
3. Add version control
4. Document package elements and test code
5. Check for errors
6. Build and install
7. Distribute/release
8. Maintain: develop new features and debug
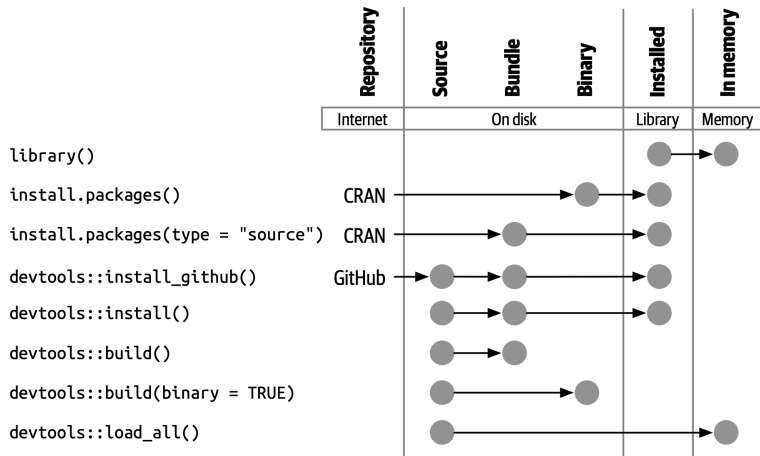
Package states and related functions

Figure 4: source: https://r-pkgs.org/structure.html#fig-installation

Available tools for package creation

- RStudio
- `devtools`: useful functions detailed, wrapper for other useful packages[1]
- `roxygen2`: exported objects documentation
- `usethis`: automates repetitive tasks for package development
- `testthat`: code tests
- Version control (Git/GitHub)
- Continuous integration

---

[1] remotes, pkgbuild, pkgload, rcmdcheck, revdepcheck, sessioninfo, usethis, testthat, and roxygen2

Supplementary exercices

- Add a package description in the DESCRIPTION file
- Add tests
- Modify the documentation with roxygen2 comments
    - Add examples
    - Add cross-references with @seealso tag and/or the [fun()] syntax
- Create a vignette
    - run usethis::use_vignette("my-vignette") to initiate a vignette
    - add a sessionInfo()[1] at the end of your vignette
- Create a website with GitHub and pkgdown (start with usethis::use_pkgdown())
- ...

---

[1]prints version information about R and attached or loaded packages.

UCLouvain

We hope this training met your expectations. Now it's up to you to continue to apply the various notions you have been taught in order to make them last...

If you have any questions or require more information about our training offer, please do not hesitate to contact us: smcs-stat@uclouvain.be – www.uclouvain.be/smcs

To keep up to date with all our news, please follow us on Linked**in**: https://be.linkedin.com/company/smcs-uclouvain

Follow us on **Linked** **in**.

Scan me
to keep up
with our
latest news