R_Packages

Ryan Womack

2023-10-16

Table of contents

1 The R packages book, by Wickham and Bryan

This is a brief introduction to R package creation, taking as its text the book R Packages, 2nd edition by Hadley Wickham and Jennifer Bryan. This book is freely available on the web, so please consult for further details of topics that are only outlined here. The second edition has been finalized in June 2023.

1.1 Install and load packages

The fundamental packages required to create an R package are *devtools*, *roxygen2*, and *testthat*. Please install these with the following commands if they are not already available on your system. These will not install by default if you just attempt to run this R markdown file.

We will use the *pak* package for installation as a more complete approach to package management. Replace the pkg commands with *install.packages()* versions if you prefer.

```
install.packages("pak", dependencies=TRUE)
library(pak)
pkg_install("devtools")
pkg_install("roxygen2")
pkg_install("testthat")

devtools::session_info()
```

Let's load those libraries now.

```
library(devtools)
```

```
Loading required package: usethis
```

```
library(roxygen2)
library(testthat)
```

```
Attaching package: 'testthat'
```

```
The following object is masked from 'package:devtools':

    test_file
```

We can check that we're running recent enough versions of our software with the *packageVersion* command for indivual packages, or *session_info* for our entire setup.

```
packageVersion("devtools")
```

```
[1] '2.4.5'
```

```r
devtools::session_info()
```

```
- Session info -------------------------------------------------------------
 setting  value
 version  R version 4.2.2 Patched (2022-11-10 r83330)
 os       Debian GNU/Linux 12 (bookworm)
 system   x86_64, linux-gnu
 ui       X11
 language (EN)
 collate  en_US.UTF-8
 ctype    en_US.UTF-8
 tz       US/Eastern
 date     2023-10-17
 pandoc   3.1.1 @ /usr/lib/rstudio/resources/app/bin/quarto/bin/tools/ (via rmarkdown)

- Packages -----------------------------------------------------------------
 package     * version date (UTC) lib source
 brio          1.1.3   2021-11-30 [2] CRAN (R 4.2.2)
 cachem        1.0.8   2023-05-01 [2] CRAN (R 4.2.2)
 callr         3.7.3   2022-11-02 [2] CRAN (R 4.2.2)
 cli           3.6.1   2023-03-23 [2] CRAN (R 4.2.2)
 crayon        1.5.2   2022-09-29 [2] CRAN (R 4.2.2)
 devtools    * 2.4.5   2022-10-11 [2] CRAN (R 4.2.2)
 digest        0.6.33  2023-07-07 [2] CRAN (R 4.2.2)
 ellipsis      0.3.2   2021-04-29 [2] CRAN (R 4.2.2)
 evaluate      0.22    2023-09-29 [2] CRAN (R 4.2.2)
 fastmap       1.1.1   2023-02-24 [2] CRAN (R 4.2.2)
 fs            1.6.3   2023-07-20 [2] CRAN (R 4.2.2)
 glue          1.6.2   2022-02-24 [2] CRAN (R 4.2.2)
 htmltools     0.5.6.1 2023-10-06 [2] CRAN (R 4.2.2)
 htmlwidgets   1.6.2   2023-03-17 [2] CRAN (R 4.2.2)
 httpuv        1.6.11  2023-05-11 [2] CRAN (R 4.2.2)
 jsonlite      1.8.7   2023-06-29 [2] CRAN (R 4.2.2)
 knitr         1.44    2023-09-11 [2] CRAN (R 4.2.2)
 later         1.3.1   2023-05-02 [2] CRAN (R 4.2.2)
 lifecycle     1.0.3   2022-10-07 [2] CRAN (R 4.2.2)
 magrittr      2.0.3   2022-03-30 [2] CRAN (R 4.2.2)
 memoise       2.0.1   2021-11-26 [2] CRAN (R 4.2.2)
 mime          0.12    2021-09-28 [2] CRAN (R 4.2.2)
 miniUI        0.1.1.1 2018-05-18 [2] CRAN (R 4.2.2)
 pkgbuild      1.4.2   2023-06-26 [2] CRAN (R 4.2.2)
 pkgload       1.3.3   2023-09-22 [2] CRAN (R 4.2.2)
 prettyunits   1.2.0   2023-09-24 [2] CRAN (R 4.2.2)
 processx      3.8.2   2023-06-30 [2] CRAN (R 4.2.2)
```

```
profvis       0.3.8   2023-05-02 [2] CRAN (R 4.2.2)
promises      1.2.1   2023-08-10 [2] CRAN (R 4.2.2)
ps            1.7.5   2023-04-18 [2] CRAN (R 4.2.2)
purrr         1.0.2   2023-08-10 [2] CRAN (R 4.2.2)
R6            2.5.1   2021-08-19 [2] CRAN (R 4.2.2)
Rcpp          1.0.11  2023-07-06 [2] CRAN (R 4.2.2)
remotes       2.4.2.1 2023-07-18 [2] CRAN (R 4.2.2)
rlang         1.1.1   2023-04-28 [2] CRAN (R 4.2.2)
rmarkdown     2.25    2023-09-18 [2] CRAN (R 4.2.2)
roxygen2    * 7.2.3   2022-12-08 [2] CRAN (R 4.2.2)
rstudioapi    0.15.0  2023-07-07 [2] CRAN (R 4.2.2)
sessioninfo   1.2.2   2021-12-06 [2] CRAN (R 4.2.2)
shiny         1.7.5.1 2023-10-14 [2] CRAN (R 4.2.2)
stringi       1.7.12  2023-01-11 [2] CRAN (R 4.2.2)
stringr       1.5.0   2022-12-02 [2] CRAN (R 4.2.2)
testthat    * 3.2.0   2023-10-06 [2] CRAN (R 4.2.2)
urlchecker    1.0.1   2021-11-30 [2] CRAN (R 4.2.2)
usethis     * 2.2.2   2023-07-06 [2] CRAN (R 4.2.2)
vctrs         0.6.4   2023-10-12 [2] CRAN (R 4.2.2)
xfun          0.40    2023-08-09 [2] CRAN (R 4.2.2)
xml2          1.3.5   2023-07-06 [2] CRAN (R 4.2.2)
xtable        1.8-4   2019-04-21 [2] CRAN (R 4.2.2)
yaml          2.3.7   2023-01-23 [2] CRAN (R 4.2.2)

[1] /home/ryan/R/x86_64-pc-linux-gnu-library/4.2
[2] /usr/local/lib/R/site-library
[3] /usr/lib/R/site-library
[4] /usr/lib/R/library
```

---

2 Outline of the creation of a little package

We'll follow along with Chapter 1 of the R Packages book and walk through the creation of a little package, even simpler than the "toy package" presented in the text.

This will enable us to review the fundamental features of a typical package:

- functions
- version control
- documentation (*roxygen2*)
- testing (*testthat*)
- creation of a README.Rmd file

After we do this for our little package, we'll coverage some additional details relating to the steps above.

2.1 Caution!

There is a conflict when using Rmarkdown to perform some of the steps below. We're presenting the code in this .Rmd file since it allows us to insert explanatory text. But you may be better off running the R_Packages.R version of this code, since that version is more straightforward. Using Rmd causes some switching back and forth between the Rmd project directory and the package directory we're creating. Just a caution!

2.2 Create_package function

We call *create_package* to initiate a package. We want to start this in its own fresh directory, not a pre-existing project or git repository. The *create_package* function will set up the necessary folder structure for a package. Please *EDIT* the contents of the command below to correspond to your computer's file system. This is the one place in the code where you'll have to modify it. Note that to be a valid package name and to be allowed on CRAN, the package name should:

- Contain only ASCII letters, numbers, and '.'
- Have at least two characters
- Start with a letter
- Not end with '.'

```
create_package("/home/ryan/R/littlePackage")
```

```
v Setting active project to '/home/ryan/R/littlePackage'
v Leaving 'DESCRIPTION' unchanged
```

```
Package: littlePackage
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)
Description: What the package does (one paragraph).
License: `use_mit_license()`, `use_gpl3_license()` or friends to pick a
    license
Encoding: UTF-8
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.2.3
```

```
v Leaving 'NAMESPACE' unchanged
v Setting active project to '<no active project>'
```

This will launch a new window. To be able to continue using our script here, we have to reopen our code (here you could just use the .R script if you want to run two RStudio sessions - one for reading the guidance and one for the package). We also have to reload our packages since this is a new session.

```
library(devtools)
library(roxygen2)
library(testthat)
```

The "dot files" beginning with a period (.) are used to store a history of the R session and to tell R and git to ignore certain files. Generally we can leave these files as is and let R worry about how to handle them. You shouldn't need to modify these files in most circumstances.

Likewise, the .Rproj file is usually left unmodified. This helps RStudio manage the package folder as a project.

The NAMESPACE is also a file we won't edit. It is used to keep track of relations between functions that your package will use, but you can let RStudio handle this.

One of the two locations that we *WILL* edit are the *DESCRIPTION*, which is a structured way of providing information about your package. This is what you will see when you look up the function in the R help system, or what would be displayed if your package makes it onto CRAN. For example, try typing ?testthat to see the description of that package.

The other is the *R* folder. This is the folder that we will put our functions into. Optionally we could add a *data* folder as well, if we wanted to distribute data via our package.

It is quite convenient to use *create_package* to take care of all of this for us.

2.3 Git

We won't go into any details about *git* or *github* usage, except to note that it is common and desirable to use this form of version control for your work, and also for sharing and collaboration. To initiate the use of git in any R project (not just packages), use the command *use_git*. To be sure we're committing the right directory, we use *proj_set* to navigate to the directory for our package first. Select the positive option when asked if you want to commit your files. For the purpose of our demo, you don't have to restart RStudio at the prompt here, but if you want to use git for real, you should. If your RStudio resets again at this stage, you'll have to do *proj_set* and reload the packages in lines 82-84 again.

```
proj_set("/home/ryan/R/littlePackage/")
```

```
v Setting active project to '/home/ryan/R/littlePackage'
```

```
use_git()
```

2.4 Writing a function

We are going to write a very basic function, just as an example. Keep in mind that you can write functions to accomplish almost any task you want to in R, and write as many of them as you want. Many packages have been born out of the gradual accumulation of useful functions. Our function will be called *funkyadd* and it just adds a little modification to addition.

```
funkyadd <- function(x,y)
  {
  x+y+1
}
```

Just for fun, let's also create a *randomadd* function that introduces true uncertainty into the addition process.

```
randomadd <- function(x,y)
  {
  x+y+round(rnorm(1,0,3),digits=0)
}
```

We can see these functions in action, attempting to add 6 and 9.

```
funkyadd(6,9)
```

```
[1] 16
```

```
randomadd(6,9)
```

```
[1] 17
```

We're not here to talk about functions, but about how functions are used in packages, so let's move on from these examples. We use *use_r* to add functions to our R directory for the our little package, as follows:

```
use_r("funkyadd")
```

```
* Edit 'R/funkyadd.R'
```

```
use_r("randomadd")
```

```
* Edit 'R/randomadd.R'
```

Copy and paste the function definitons into the editor window that pops up. Namely, lines 124-127 for *funkyadd* and lines 132-135 for *randomadd*.

Now we run *load_all* to bring our functions as we've defined them into the R workspace. Note that we remove the previous manually created functions just to show that *load_all* is working properly. Note that the *library* command only works for installed packages. Since we haven't finished with our package yet, we need to use *load_all* instead. We also reconfirm that we're in the correct project directory. Then we can check that the functions are in our workspace, now provided directly by the package definitions. [Note that at this point we're no longer going to print output from our commands in this summary text, due to a conflict between the package directory and our home directory.]

```
rm(funkyadd)
rm(randomadd)
setwd("/home/ryan/R/littlePackage/")
proj_set("/home/ryan/R/littlePackage/")
load_all()
funkyadd(3,5)
randomadd(3,5)
```

In the actual development of a package, it is recommended to commit your changes using git as you go along, but we'll omit those steps for clarity of exposition here.

2.5 Checking your work

R has a function to look for any errors in your package. In the terminal, you could type *R CMD check*. Within R or Rstudio, use *check*

```
check()
```

We get a lot of output and a useful note and a warning in this case.

2.6 Filling in the details

Edit the DESCRIPTION file using RStudio. Just insert your own name and descriptive information in the fields and save.

Run *use_mit_license* to insert a complete current license for your package. Note the appearance of the LICENSE files after this. We could also use GPL or other licenses. Use the *document* command to make it finito.

```
use_mit_license()
```

If you use RStudio, open R/funkyadd.R in the source editor and put the cursor somewhere in the *funkyadd* function definition. Now select *Code > Insert roxygen skeleton* from the RStudio menu. A very special comment should appear above your function, in which each line begins with #'. RStudio only inserts a barebones template, so you will need to edit it to add descriptive information after the #' @____ characters.

Now we run the *document* command to generate Rdocumentation format files from our R commands. We can repeat this process for *randomadd*. Note that we could also create Rdocumentation (.Rd) files in a text editor, but the roxygen/document process saves us from learning a new set of markup tags.

```
document()
?funkyadd
```

Now we can get help on *funkyadd* with the usual ?funkyadd syntax. The NAMESPACE file has also been updated by the *document* command.

We can now *check* and *install* the package to include it in our R space. We can now load it with a *library* command, although you may still want to *load_all* to be safe. Our functions should work now.

```
check()
install()
library(littlePackage)
load_all()
randomadd(3,6)
```

We can test our package systematically by loading *use_testthat* and then running tests using *use_test*. Note that we have to create our own tests to run them. These might check, for example, that the results of certain functions fall into the values we expect them to take. The *use_test* function will insert properly named R files into a "tests" directory, but it is up to you to edit them into something sensible. Once tests are created, you can run them all with the *test* command.

```
use_testthat()
use_test("funkyadd")
test()
```

2.7 Some additional useful commands

We can require the use of certain packages with the *use_package* command.

We can connect our project to a specific Github repository using the *use_github* command.

For usage on Github, we need a more complete README file. The *use_readme_rmd* command will set this up for us. This creates a README.Rmd file that is structured for typical R package usage, along with a process for generating a Github-friendly README.md file as well. Just use the *build_readme* command to render the .md formatted file.

These steps are highly recommended for a working package, but we won't try to reproduce them in this short introduction.

2.8 Finishing up the little package

A final *check* and *install* once you are satisfied with all the edits on your package will finalize, rebuild, and install the package properly.

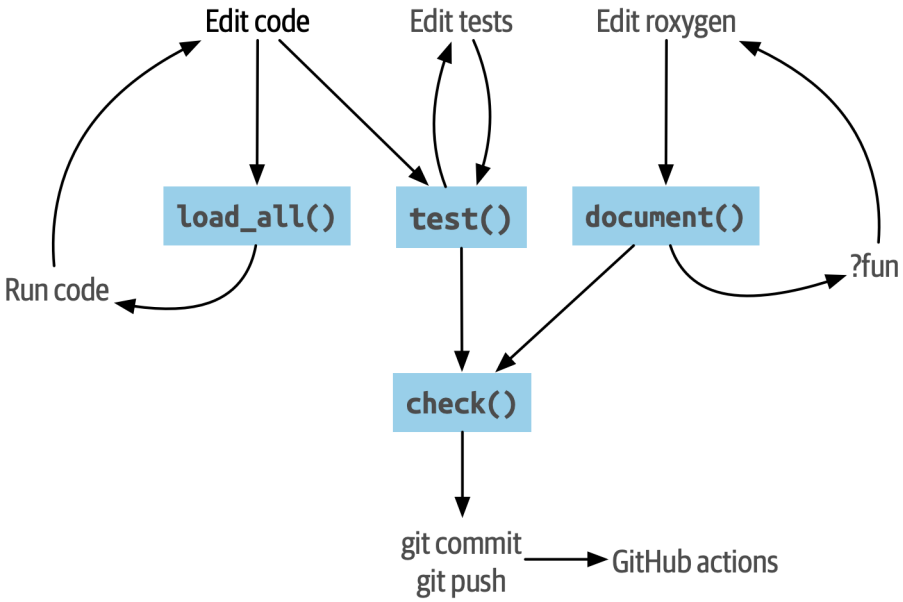
```
check()
install()
```

2.9 Review of command sequence

We used the following commands to step through the creation, editing, and testing of our package.

- create_package()
- use_git()
- use_r()
- load_all()
- check()
- use_mit_license()
- document()
- check() [again]
- edit the DESCRIPTION using RStudio
- install()
- use_testthat()
- use_package()
- use_github()

- use_readme_rmd()
- check() [last time]
- install() [last time]



Quoting the R Packages book:

Here is a review of the key functions you've seen here, organized roughly by their role in the development process.

These functions setup parts of the package and are typically called once per package:

- create_package()
- use_git()
- use_mit_license()
- use_testthat()
- use_github()
- use_readme_rmd()

You will call these functions on a regular basis, as you add functions and tests or take on dependencies:

- use_r()
- use_test()
- use_package()

You will call these functions multiple times per day or per hour, during development:

- load_all()
- document()
- test()
- check()

3 Additional miscellaneous highlights from R Packages

The available package has a function called *available()* that helps you evaluate a potential package name from many angles

Here are the most common gotchas that trip many of us up at first:

Package code requires new ways of working with functions in other packages. The **DESCRIPTION** file is the principal way to declare dependencies; we don't do this via *library(somepackage)*. Instead name the package in the "imports" section of DESCRIPTION file. If you want data or files to be persistently available, there are package-specific methods of storage and retrieval. You can't just put files in the package and hope for the best. It's necessary to be explicit about which functions are user-facing and which are internal helpers. By default, functions are not exported for use by others. A new level of discipline is required to ensure that code runs at the intended time (build time vs. run time) and that there are no unintended side effects.

Refer to commands in long form, e.g.

dplyr::mutate

It is natural to assume that listing a package in Imports actually "imports" the package, but this is just an unfortunate choice of name for the Imports field. The Imports field makes sure that the packages listed there are installed when your package is installed. It does not make those functions available to you, e.g. below R/, or to your user.

Every package mentioned in NAMESPACE must also be present in the Imports or Depends fields.

3.1 Code style [extracts from R Packages book]

We recommend following the tidyverse style guide, which goes into much more detail than we can here.

Although the style guide explains the "what" and the "why", another important decision is how to enforce a specific code style. For this we recommend the styler package; its default behaviour enforces the tidyverse style guide.

There are some functions that modify global settings that you should never use because there are better alternatives:

Don't use *library()* or *require()*. These modify the search path, affecting what functions are available from the global environment.Instead, you should use the DESCRIPTION to specify your package's requirements, as described in Chapter 9. This also makes sure those packages are installed when your package is installed.

Never use *source()* to load code from a file. *source()* modifies the current environment, inserting the results of executing the code. There is no reason to use *source()* inside your package, i.e. in a file below R/. Sometimes people *source()* files below R/ during package

development, but as we've explained in Section 4.4 and Section 6.2, *load_all()* is a much better way to load your current code for exploration. If you're using *source()* to create a dataset, it is better to use the methods in Chapter 7 for including data in a package.

Here is a non-exhaustive list of other functions that should be used with caution:

options() par() setwd() Sys.setenv() Sys.setlocale() set.seed() (or anything that changes the state of the random number generator)

We usually manage state using the withr package we need to "Restore the state" to what we left it . These functions do this automatically for us.

For example, change working directory using *with_dir()*

3.2 Testing [extracts from R Packages book]

testthat function is most commonly used

to setup your package to use *testthat*, run:

usethis::use_testthat(3)

Can create in /tests/testthat directory, any tests to run and testthat.R file to run the tests

for example:

The *foofy()* function (and its friends and helpers) should be defined in R/foofy.R and their tests should live in tests/testthat/test-foofy.R.

functions *use_r() / use_test()* are handy for initially creating these file pairs and, later, for shifting your attention from one to the other.

When *use_test()* creates a new test file, it inserts an example test.

mockery is also useful in this context.

Github Actions are recommended as a way to improve the checking and testing process.

3.3 Documentation [extracts from R Packages book]

In the devtools ecosystem, we don't edit .Rd files directly with our bare hands. Instead, we include specially formatted "roxygen comments" above the source code for each function1. Then we use the roxygen2 package to generate the .Rd files from these special comments2 . There are a few advantages to using roxygen2 :

- Code and documentation are co-located. When you modify your code, it's easy to remember to also update your documentation.
- You can use markdown, rather than having to learn a one-off markup language that only applies to .Rd files. In addition to formatting, the automatic hyperlinking functionality makes it much, much easier to create richly linked documentation.

- There's a lot of .Rd boilerplate that's automated away.
- roxygen2 provides a number of tools for sharing content across documentation topics and even between topics and vignettes.

Roxygen comment lines always start with #' , the usual # for a comment, followed immediately by a single quote '

To summarize, there are four steps in the basic roxygen2 workflow:

- Add roxygen2 comments to your .R files.
- Run devtools::document() or press Ctrl/Cmd + Shift + D to convert roxygen2 comments to .Rd files.
- Preview documentation with ?function.
- Rinse and repeat until the documentation looks the way you want.

The title is taken from the first sentence. It should be written in sentence case, not end in a full stop, and be followed by a blank line. The title is shown in various function indexes (e.g. help(package = "somepackage")) and is what the user will usually see when browsing multiple functions.

The description is taken from the next paragraph. It's shown at the top of documentation and should briefly describe the most important features of the function.

Additional details are anything after the description. Details are optional, but can be any length so are useful if you want to dig deep into some important aspect of the function. Note that, even though the details come right after the description in the introduction, they appear much later in rendered documentation.

The book also addresses **vignettes**, which allow a more in-depth view of your package functionality

pkgdown can help you build a website for your package easily and easily manage links.

Should also have a README.md file for basic explanation (which can be generated from an .Rmd file) and perhaps a NEWS.md file

There is even guidance on producing a logo and a hex sticker

3.4 Licensing [extracts from R Packages book]

If you want a permissive license so people can use your code with minimal restrictions, choose the MIT license with *use_mit_license()*.

If you want a copyleft license so that all derivatives and bundles of your code are also open source, choose the GPLv3 license with *use_gpl_license()*.

If your package primarily contains data, not code, and you want minimal restrictions, choose the CC0 license with *use_cc0_license()*. Or if you want to require attribution when your data is used, choose the CC BY license by calling *use_ccby_license()*.

If you don't want to make your code open source, call *use_proprietary_license()*. Such packages can not be distributed by CRAN.

We highly recommend https://choosealicense.com,

For more details about licensing R packages, we recommend Licensing R by Colin Fay

3.5 Data [extracts from R Packages book]

LazyData is relevant if your package makes data available to the user. If you specify LazyData: true, the datasets are lazy-loaded, which makes them more immediately available, i.e. users don't have to use *data()*. The addition of LazyData: true is handled automatically by *usethis::use_data()*

If you want to store R objects and make them available to the user, put them in data/ Each file should be an .rda file created by *save()* containing a single R object, with the same name as the file. The easiest way to achieve this is to use *usethis::use_data()*."

It is also common for data packages to provide, e.g., a csv version of the package data that is also provided as an R object. This data is placed in the *inst/extdata* directory so that it is visible to end users of the package in the *extdata* directory.

Often, the data you include in data/ is a cleaned up version of raw data you've gathered from elsewhere. We highly recommend taking the time to include the code used to do this in the source version of your package. This makes it easy for you to update or reproduce your version of the data. This data-creating script is also a natural place to leave comments about important properties of the data, i.e. which features are important for downstream usage in package documentation. This data should be kept in a *data-raw* directory, best handling by the *usethis::use_data_raw()* function.

Package data submitted to CRAN should be less than 1MB or you will need to argue for an exemption. Also consider compression.

Use the usual method to document your dataset.

There are two roxygen tags that are especially important for documenting datasets:

- @format gives an overview of the dataset. For data frames, you should include a definition list that describes each variable. It's usually a good idea to describe variables' units here.
- @source provides details of where you got the data, often a URL.

Never @export a data set.

Other uses for data are discussed in Chapter 7 of R packages