Package Development

Statistical Computing, STA3005

Tuesday Mar 13, 2023

Last chapter: Functions

- Function: formal encapsulation of a block of code; generally makes your code easier to understand, to work with, and to modify
- Functions are absolutely critical for writing (good) code for medium or large projects
- A function's structure consists of three main parts: inputs, body, and output
- R allows the function designer to specify default values for any of the inputs
- R doesn't allow the designer to return multiple outputs, but can return a list
- Side effects are things that happen as a result of a function call, but that aren't returned as an output
- Top-down design means breaking a big task into small parts, implementing each of these parts, and then putting them together

Part I: *Overview*

Why do we develop R packages?

Packages are the fundamental units of sharable R code. A package bundles **code**, **data**, **documentation** and **tests** together, and is easy to share with others. From now on, there are close to 20,000 R packages available on CRAN. For you, package development helps

- Understand others' packages more quickly and deeply
- Share your own methods in the future

Philosophy of package development

Anything can be automated should be automated.

Let's focus on the contents (functions) rather than structure by using `devtools` package. Some functions rely on a C complier and command-line tools (depending on your operation system)

- `Rtools` for Windows
- `XCode` for Mac OS
- `r-base-dev` for Ubuntu

Create an R package

- Drop-down menu: File→New Project→New Directory → R package
- Run `devtools::create("path/to/package/pkgname")`

Notice that `::` allows us to call functions in an unloaded package.

```
devtools::create("WordTab")
```

Rstudio projects

The above two creating methods will create an isolated environment for package development, called a project.

- Code running in one project does not affect any other projects

- Easy to continue by double-click the `.Rporj` file

Package state

There are five states for a package across its life cycle:

design
- source: the development version that lives on your computer.
- bundled: compress the source package into a single `.tar.gz` file
- binary: a single file can be installed on the computer without package development tools

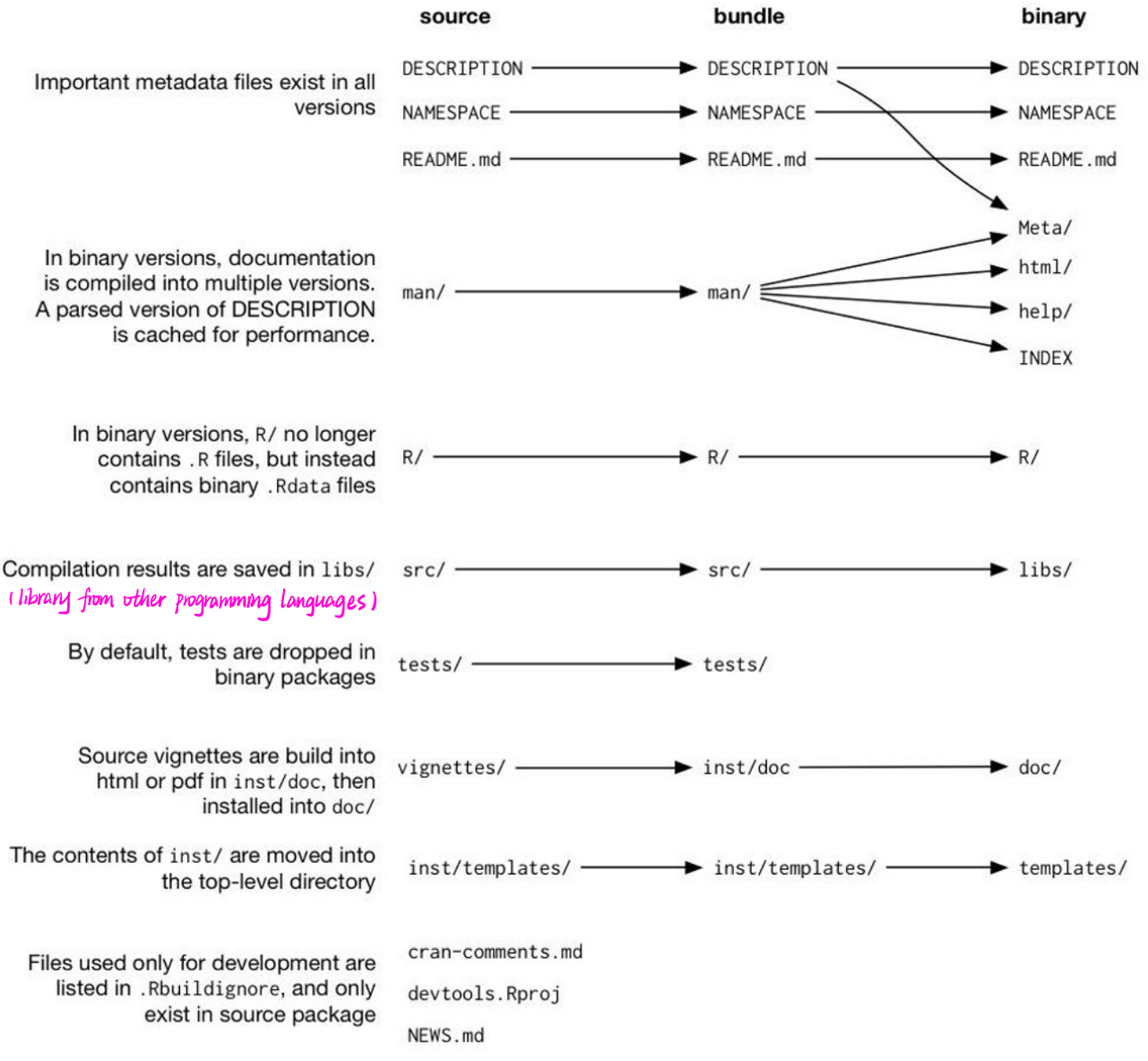
use
- installed: a binary package that has been decompressed into a package library
- in-memory: a package loaded into memory

Composition of source, bundled and binary packages

In package development, we work on the source package and then compress the source file as a bundled package by

```
devtools::build()
```

Finally, your submission should be a bundled file.

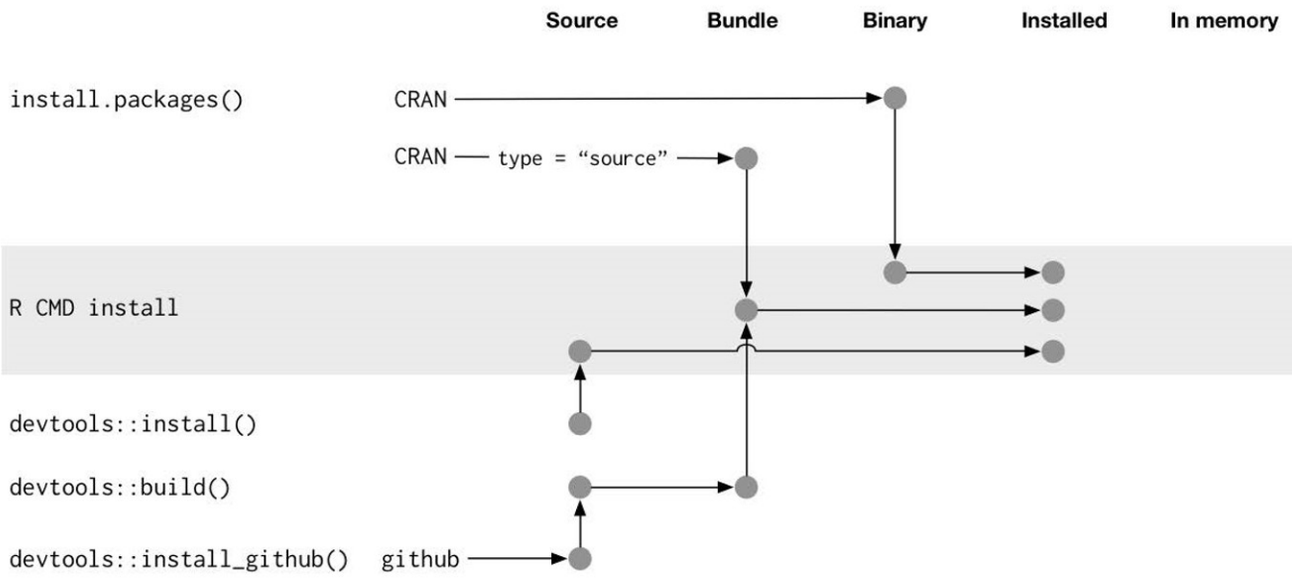


Source, bundled and binary packages

Package installation

There are different ways to install a package.

- The tool that powers all package installation is the command-line tool `R CMD INSTALL`, which can install a source, bundled, or binary package.
- `devtools` functions provide wrappers that allow you to access installation from R console rather than the command line, including `devtools::install()`, `devtools::build()` and `devtools::install_github()`
- `install.packges()` allows you to install packages from CRAN



Package installation

Part II: *R code*

R code

All R code goes in `R/` directory. Filenames should be meaningful and end in `.R`:

```
# Good
fit_models.R
utility_functions.R

# Bad
foo.r
stuff.r
```

Create R file for objects, the marjority of which will be functions and some for datasets (More details in the following).

Do not change R landscape

A big difference between code in scripts and packages is that other people are going to use your package, and they're going to use it in situations that you never imagined.

Thus, you should avoid changing the global settings

- `library()`: modify the search path `search()`
- `options()`: modify the global option
- `setwd()`: modify the working directory
- `par()`: modify the graphics parameters

If you have to, recover by `on.exit()`

```
old_par <- par(mfrow = c(1,2)) 会返回原本的 parameters
on.exit(par(old_par), add = TRUE)
  function 使用结束后，会恢复原本的 parameters
```

```
old_dir <- setwd(tempdir())
on.exit(setwd(old_dir), add = TRUE)
```

Part III: *Metadata or DESCRIPTION*

DESCRIPTION

DESCRIPTION stores important metadata about your package. Every package must have a DESCRIPTION. In fact, it's the defining feature of a package.

DESCRIPTION file uses the Debian control format. Each line consists of a *field* name and a value separated by a colon. When values spans multiple lines, they need to be indented:

```
Package: WordTab
Type: Package
Title: What the Package Does (Title Case)
Version: 0.1.0
Author: Zhaorui Dong <zhaoruidong@link.cuhk.edu.cn>, Fangda Song <songfangda@cuhk.edu.cn>
Maintainer: Fangda Song <songfangda@cuhk.edu.cn>
Description: More about what it does (maybe more than one line)
    Use four spaces when indenting paragraphs within the Description.
License: What license is it under?
Encoding: UTF-8
LazyData: true
```

Dependencies: What does your package need?

You should list the packages necessary for your package to work using `Imports` and `Suggests:` fields.

`Imports` field:

- Must be present for using your package
- Installed automatically
- Added by `devtools::use_package("pkg")`

The following lines indicate that your package needs both `ggvis` and `dplyr` to work:

```
Imports:
  dplyr,
  ggvis
```

`Suggests` field:

- Only a partion of your package depends on the package
- Not installed automatically
- Added by `devtools::use_package("pkg", "Suggests")`

The following lines indicate that while your package can take advantage of `ggvis` and `dplyr`, there are not required to make it work:

```
Suggests:
  dplyr,
  ggvis
```

Versioning: `pkg (>=1.3.0)`

Title and Description fields

`Title` field: one-line description up to 65 characters

`Description` field: one-paragraph description

Ex: `Title` and `Description` of `ggplot2`

```
Title: Create Elegant Data Visualisations Using the Grammar of Graphics
Description: A system for 'declaratively' creating graphics, based on "The
    Grammar of Graphics". You provide the data, tell 'ggplot2' how to map
    variables to aesthetics, what graphical primitives to use, and it
    takes care of the details.
```

Author, version and license fields

`Author` field:

```
AuthorName1 <EmailAddress@link.cuhk.edu.cn>, AuthorName2 <EmailAddress@link.cuhk.edu.cn>
```

`Maintainer` field:

```
Maintainer <EmailAddress@link.cuhk.edu.cn>
```

- At least one authors and only one maintainer

`Version` field: `X.Y.Z` (such as 1.0.5)

- `X`: Marjor number
- `Y`: Minor number
- `Z`: Patch number
- `X` >= 1: Released version
- `0.9.Z`: In-developing version

`License` field: A standard abbrevation open source license, like GPL-2, or a pointer to a file containing more information, `file LICENSE`.

(general public license)

- `License: GPL-2`

Part IV: *Object documentation and Namespace*

Documentation is one of the most important aspects of a good package. Without it, users don't know how to use your package. It is also useful for "future you" and for developers extending your package.

Recall that we can invoke object documentation by `?` or `help()`. It works like a **dictionary**:

- Explain the meaning of an object when you give its name
- Cannot help you find the object you need to solve a given problem (Solved by **vignettes**)

Traditional way of documenting: write `.Rd` files in the `man/` directory

Instead of writing `.Rd` files by hand, we use `roxygen2` package to generate `.Rd` files automatically

- R code and documentation are linked
- `roxygen2` dynamically inspects the objects it documents
- it unifies the documentation of different objects

Documentation workflow

- Add roxygen comments to your `.R` files
- Run `devtools::document()` to convert roxygen comments to `.Rd` files
- Preview documentation with `?`

- Modify and repeat until the documentation looks the way you want

Roxygen comments

- Syntax: start with `#'`
- Block: the roxygen lines preceding a function
- Multiple tags in a block: `@tagName details`

Heading of a block: - First sentence: title of documentation - Second paragraph: description - Third and subsequent paragraphs: details

Document functions

`@param name description` - All arguments should be documented - Multiple similar arguments in one place

`@examples` - Executable R code show how to use this function - Very important: many people look at the example first

`@return description` - Describe the output of this function

Document datasets and packages

Document data in `data/` directory is like documenting a function with additional tags, such as `@format`.

```
#' Prices of over 50,000 round cut diamonds  (title)
#'
#' A dataset containing the prices and other attributes of almost 54,000
#'  diamonds. The variables are as follows:  (description)
#'
#' @format A data frame with 53940 rows and 10 variables: (details)
#' \describe{
#'   \item{price}{price in US dollars ($326--$18,823)}
#'   \item{carat}{weight of the diamond (0.2--5.01)}
#'   \item{cut}{quality of the cut (Fair, Good, Very Good, Premium, Ideal)}
#'   \item{color}{diamond colour, from D (best) to J (worst)}
#'   \item{clarity}{a measurement of how clear the diamond is (I1 (worst), SI2,
#'     SI1, VS2, VS1, VVS2, VVS1, IF (best))}
#'   \item{x}{length in mm (0--10.74)}
#'   \item{y}{width in mm (0--58.9)}
#'   \item{z}{depth in mm (0--31.8)}
#'   \item{depth}{total depth percentage = z / mean(x, y) = 2 * z / (x + y) (43--79)}
#'   \item{table}{width of top of diamond relative to widest point (43--95)}
#' }
"diamonds"
```

Document the whole R package: use to describe the most important components of your package.

- Create a `<package-name>.R` file in `man/`
- Document `NULL` and mutually label it with
- `@section` tag: divide up a page into useful categories

Ex:

```
#' foo: A package for computing some statistics
#'
#' The foo package provides three categories of important functions:
#' foo, bar and baz
#'
#' @section Foo functions:
#' The foo functions ...
#'
```

```
#' @docType package
#' @name foo
NULL
```

More `roxygen` syntax

Special characters

- `@`, `%` and `\` replaced by `@@`, `\%` and `\\`, respectively

Character formatting:

- `\emph{}` for *italics*
- `\strong{}` for **bold**
- `\code{}` for inline codes

Links:

- To other functions: `\code{\link{function}}`
- To websites: `\url{http:\\rstudio.com}`

Lists, equations and tables

- Ordered lists

```
#' \enumerate{
#'   \item First item
#'   \item Second item
#' }
```

- Unordered lists

```
#' \itemize{
#'   \item First item
#'   \item Second item
#' }
```

- Definition lists

```
#' \describe{
#'   \item{One}{First item}
#'   \item{Two}{Second item}
#' }
```

Namespace

Namespace is the search path of the package environment and claims avaiable functions in the package

- `imports` : like `library()` in global environment (不能使用 library() 改变 global env)
- `exports` : indicating visible functions in your package

Automatically generate by `roxygen` package by the following tags

Import:

- `#' @importFrom pkg fun` : import a single function `fun` from pacakge `pkg`
- `#' @import pkg` : import all functions in package `pkg`

Export:

- `#' @export` : export the proceeding function

`import()` in NAMESPACE and `Imports` field in DESCRIPTION

Tag	`import()` or `importFrom()`	Imports field
Location	NAMESPACE	DESCRIPTION
Similarity	indicate package dependencies	
Function ~~Location~~	Import dependent packages into NAMESPACE (search path of package environment)	Automatically install dependent packages when your package is installed

Part V: *Automated checking*

Automated checking

Command-line `R CMD check`:

- automatically check your code for common problems
- a command can be run from the terminal
- `system("R CMD check ./")`

`devtools::check()`

- automatically run `devtools::document()`
- automatically bundle the package before checking

Basic requirement to release you R package

- Your submission must pass `R CMD check` with no errors and warnings

Three kinds of messages

`ERROR`: Severe problems that you should fix `WARNING`: Likely problems that you must solve before release
`NOTE`: Mild problems

Workflow:

- Run `devtools::check()` in console
- Fix the first problem
- Repeat until there is no more problem

Part VI: *Vignettes*

Vignette: Long-form documentation

A vignette is a long-form guide to your package. It looks like a book chapter or an academic paper

- Describe the problem that your package is designed to solve
- Illustrate how to solve it
- Divide functions into useful categories
- Demonstrate how to coordinate multiple functions

Browse the installed vignettes in `inst/doc` directory

```
browseVignettes("pkgName")
```

Each vignette provides three things:

- the original source file

- a readable HTML page or PDF
- a file of R code

Workflow of building vignettes

To generate a vignette, we use the R markdown vigenette engine provided by the `knitr` package.

- Create `vignettes/` directory by `devtools::use_vignette("VignetteName")`
- Modify the vignette under Rmarkdown syntax
- Press Ctrl/Cmd-Shift-K to compile the vignette and preview the output

Composition of Vignettes source code

Vignettes source code can be regarded as a Rmarkdown file, comprising of three parts:

- Initial metadata block
- Markdown for formatting text
- Knitr for intermingling text, code and results

Metadata

Metadata is similar to the syntax of DESCRIPTION

- title and author fields
- date fields: by default 2024-03-04` to insert today's date
- output and vignette field: by default

Markdown

A lightweight text-to-html markup language designed by John Gruber in 2004.

- Not as powerful as LaTeX
- Simple, easy to write, easy to read

Rmarkdown is based on Pandoc's version of Markdown. It provides extra syntax for tables, definition lists, footnotes, citations and so on

Syntax of R markdown

Sections:

- `#` for Heading 1;
- `##` for Heading 2;
- `###` for Heading 3;
- `---` or `***` for horizontal dividing line

Lists:

- `-` or `*` for unordered list
- `1.`, `2.`, `3.` for ordered list

Code:

1. inline code: between two back ticks
2. A chunk of code: between two group of triple back ticks
3. Highlight the code as the syntax of R (```{r})

More details about inline formatting and tables involved in the tutorial

Knitr

`Knitr` allows us to intermingle code, results and code by Knitr

- Run R code
- Capture the output
- Translate into formatted Markdown

These three steps are called rendering

Options of rendering

Chunk setting:

```
{r, opt1 = val1, opt2 = val2}
```

Global setting:

```
knitr::opts_chunk$set(
  opt1 = val1,
  opt2 = val2)
```

- `eval = FALSE` : not evaluate or run the code chunk
- `echo = FALSE` : not print the code chunk
- `results = "hide"` not print the output
- `warning = FALSE` and `message = FALSE` : not display warnings and messages
- `error = TRUE` : show errors and keep running
- `purl = FALSE` : not include in the R code collection of Vignettes.
- `collapse = TRUE` and `comment = "#>"` : controls how to display code output.

有多个 outputs 时会一起输出

More options are avaiable here

Development cycle

1. Debug each chunk: Ctrl/Cmd-Alt-C
2. Debug the entire document: Ctrl/Cmd-Shift-K
3. Build all vignettes: `devtools::build_vignettes()` (rarely used) `devtools::build()`
4. Get feedback from people who do not know anything about your package
5. Do 1-4 recursively

仅用这个时会一起 build vignettes

When writing a vignette, you are teaching someone how to use your package. You need to put yourself in the reader's shoes and adopt a beginner's mind.

Part VII: *Data*

Include datasets in your package

Data support compelling use cases for package functions. There are two different ways:

- Exported data: Store binary data and make it available to user, put it in `data/extdata.Rdata`
- Internal data: Store parsed data, but not make it available to the user, put it in `R/sysdata.rda`

Workflow

1. Create a data object
2. In the console, run

```
devtools::use_data(..., internal = FALSE)
```

(会储存 objects)

Document exported data

Create an R file in `R\` with the same name as the dataset and write the `roxygen` comment in the file as mentioned above.

Part VIII: *Testing*

Ensure your code does as you want

In general, we manually test all the functions by

1. Load them by `load_all()`
2. Try some testing code in the console
3. Modify your functions when unexpected results appear
4. Repeat 1-3 recursively

You may forget the testing code after a period of time

- Manual tests –> Automated tests

Automated tests

`devtools::use_testthat()` : apply testthat package for automated tests

1. Create a `test/testthat` directory
2. Add testthat to the `Suggests` file in DESCRIPTION
3. Create a `texts/testthat.R` that runs all your tests are when `R CMD check` runs

Once you have set up, the workflow turns to

1. Modify your code or tests
2. Test your package with `devtools::test()`
3. Repeat until all tests pass.

Test structure:

```
test_that(desc, code)
```

- `desc` : Test name
- `code` : Test code containing expectations
- Expectations: functions that start with `expect_`

Tests are organized hierachically： expectations are grouped into tests. Here, we take the `stringr` package as an example

```
test_that("str_length is number of characters", {
  expect_equal(str_length("a"), 1)
  expect_equal(str_length("ab"), 2)
  expect_equal(str_length("abc"), 3)
})

test_that("str_length of missing string is missing", {
  expect_equal(str_length(NA), NA_integer_)
  expect_equal(str_length(c(NA, 1)), c(NA, 1))
  expect_equal(str_length("NA"), 2)
})

test_that("str_length of factor is length of level", {
  expect_equal(str_length(factor("a")), 1)
  expect_equal(str_length(factor("ab")), 2)
  expect_equal(str_length(factor("abc")), 3)
})
```

Expectations are the finest level of testing. It makes a binary assertion about whether or not a function call does what you expect. They have a similar structure:

```
expect_equal(obj, expected)
```

- `obj` : the actual results by evaluating code
- `expected` : the expected results
- Throws an error, if two results don't agree

More expectation functions:

- `expect_match` : inspects a string
- `expect_output` : inspects printed output
- `expect_warnings` : inspects warnings

What to test

There is a trade-off. Tests make your code less likely to change inadvertently; it also can make it harder to change your code on purpose. There are some helpful points to build a test:

- Focus on external interface to your functions
- Avoid testing trivial code that you're confident
- Always write a test when you discover a bug

Summary

Philosophy: Anything can be automated should be automated.

The entire workflow of package development can be summarized as:

1. Create R package by `devtools::create()`
2. Modify partial fields in DESCRIPTION
3. Write R functions in `R/` directory
4. Write object documentation in `R` files and automatically generate `Rd` file by `roxygen`
5. Specify the dependency and visibility of functions or packages in NAMESPACE by `roxygen`
6. Write Vignettes in `vignettes/` directory
7. Include datasets in `data/` by `devtools::use_data()`
8. Write automated tests in `test/` by `devtools::use_testthat()`
9. Repeat 2-8 until passing automated checking by `devtools::check()`