# Making an R package(II)

*Canhong Wen*

## Package components

1. Code (`R/`)
2. Package metadata (`DESCRIPTION`)
3. Object documentation (`man/`)
4. Vignetters (`vignettes/`)
5. Namespaces (`NAMESPACE`)
6. Data (`data/`)
7. Compiled code (`src/`)

## R code

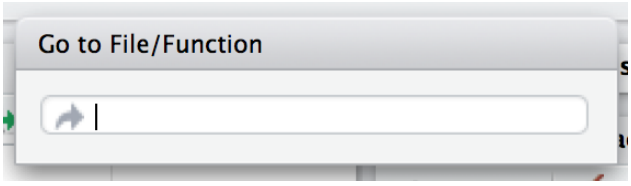
- The first practical advantage to using a package is that it's easy to re-load your code.
- You can either run `devtools::load_all()`, or in RStudio press **Ctrl/Cmd + Shift + L**, which also saves all open files, saving you a keystroke.
- This keyboard shortcut leads to a fluid development workflow:
  1. Edit an R file.
  2. Press Ctrl/Cmd + Shift + L.
  3. Explore the code in the console.
  4. Rinse and repeat.

## Organising your functions

- Don't put all functions into one file
- Don't put each function into its own separate file. (It's OK if some files only contain one function, particularly if the function is large or has a lot of documentation.).
- File names should be meaningful and end in .R.

```
# Good
fit_models.R
utility_functions.R

# Bad
foo.r
stuff.r
```

- Find a function: Press Ctrl + . then start typing the name:



## Object names

- Variable and function names should be lowercase. Use an underscore (_) to separate words within a name (reserve . for S3 methods).
- Camel case is a legitimate alternative, but be consistent!
- Generally, variable names should be nouns and function names should be verbs.

```
# Good
day_one
day_1

# Bad
first_day_of_the_month
DayOne
dayone
djm1

# Bad
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

## Spacing

- Place spaces around all infix operators (=, +, -, <-, etc.). The same rule applies when using = in function calls.
- Always put a space after a comma, and never before.

```
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Bad
average<-mean(feet/12+inches,na.rm=TRUE)
```

- There are a small exception to this rule: : and :: don't need spaces around them.

```
# Good
x <- 1:10
base::get

# Bad
x <- 1 : 10
base :: get
```

- Extra spacing (i.e., more than one space in a row) is ok if it improves alignment of equal signs or assignments (<-).

```
list(
  total = a + b + c,
  mean  = (a + b + c) / n
)
```

- Do not place spaces around code in parentheses or square brackets

```r
# Good
if (debug) do(x)
diamonds[5, ]

# Bad
if ( debug ) do(x)  # No spaces around debug
x[1,]   # Needs a space after the comma
x[1 ,]  # Space goes after comma not before
```

## Curly braces

- An opening curly brace should never go on its own line and should always be followed by a new line.
- A closing curly brace should always go on its own line, unless it's followed by else.
- Always indent the code inside curly braces.

```r
# Good

if (y < 0 && debug) {
  message("Y is negative")
}

if (y == 0) {
  log(x)
} else {
  y ^ x
}
```

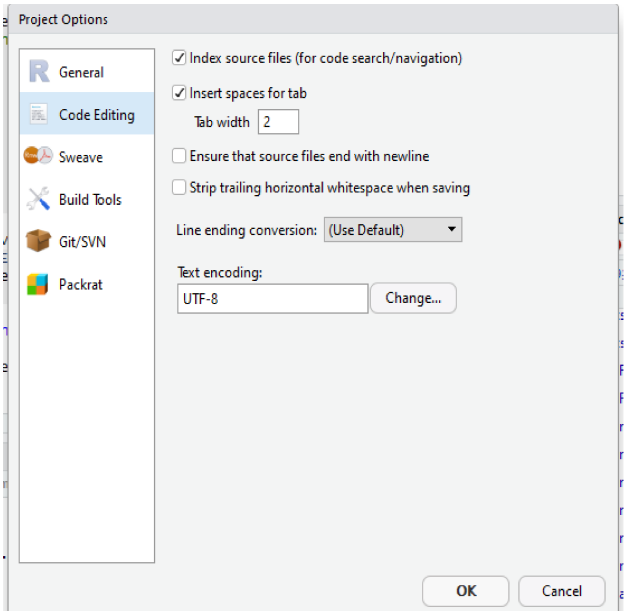
```r
# Bad

if (y < 0 && debug)
message("Y is negative")

if (y == 0) {
  log(x)
}
else {
  y ^ x
}
```

- It's ok to leave very short statements on the same line:

```r
if (y < 0 && debug) message("Y is negative")
```

## Line length and Indentation

- Limit your code to 80 characters per line
- When indenting your code, use two spaces.




- The only exception is if a function definition runs over multiple lines. In that case, indent the second line to where the definition starts:

```
long_function_name <- function(a = "a long argument",
                               b = "another argument",
                               c = "another long argument") {
  # As usual code is indented by two spaces.
}
```

## Commenting guidelines

- Each line of a comment should begin with the comment symbol and a single space: `#`.
- Use commented lines of `-` and `=` to break up your file into easily readable chunks.

```
# Load data ---------------------------

# Plot data ---------------------------
```

## Codes in scripts and packages

- R code saved in a file that you load with `source()`. So what is the difference between code in scripts and packages:
  - In a script, code is run when it is loaded.
  - In a package, code is run when it is built.
  - This means your package code should only create objects, the vast majority of which will be functions.

```
# It is ok in your script file and source it.
library(ggplot2)

show_mtcars <- function() {
```

```
  qplot(mpg, wt, data = mtcars)
}

# It won't work if you include the above code in you package foo
library(foo)
show_mtcars()
```

- The code won't work because ggplot2's `qplot()` function won't be available: `library(foo)` doesn't re-execute `library(ggplot2)`.

## Part II

*Package metadata*

### Every package must have a `DESCRIPTION`

```
Package: pkgA
Type: Package
Title: What the Package Does (Title Case)
Version: 0.1.0
Author: Who wrote it
Maintainer: The package maintainer <yourself@somewhere.net>
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
```

```
Package: pkgB
Type: Package
Title: What the Package Does in One 'Title Case' Line
Version: 1.0
Date: 2019-11-21
Author: Your Name
Maintainer: Your Name <your@email.com>
Description: One paragraph description of what the package does as one or more full sentences.
License: GPL (>= 2)
Imports: Rcpp (>= 1.0.2)
LinkingTo: Rcpp
```

```
Package: BeSS
Type: Package
Title: Best Subset Selection in Linear, Logistic and CoxPH Models
Version: 1.0.6
Date: 2019-02-19
Author: Canhong Wen, Aijun Zhang, Shijie Quan, Xueqin Wang
```

```
Maintainer: Canhong Wen <wencanhong@gmail.com>
Description: An implementation of best subset selection in generalized linear model
    and Cox proportional hazard model via the primal dual active set algorithm
    proposed by Wen, C., Zhang, A., Quan, S. and Wang, X. (2017) <arXiv:1709.06254>.
    The algorithm formulates coefficient parameters and residuals as primal and dual
    variables and utilizes efficient active set selection strategies based on the
    complementarity of the primal and dual variables.
License: GPL-3
Depends: R (>= 3.2.0)
Imports: Rcpp(>= 0.12.8), Matrix(>= 1.2-6), glmnet, survival
LinkingTo: Rcpp, RcppEigen
NeedsCompilation: yes
Packaged: 2019-02-19 15:49:51 UTC; quanshijief
Repository: CRAN
Date/Publication: 2019-02-21 07:20:07 UTC
Built: R 3.6.1; x86_64-w64-mingw32; 2019-10-26 03:39:29 UTC; windows
Archs: i386, x64
```

```
Package: ggplot2
Version: 3.2.1
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.
Authors@R: c(
    person("Hadley", "Wickham", , "hadley@rstudio.com", c("aut", "cre")),
    person("Winston", "Chang", , role = "aut"),
    person("Lionel", "Henry", , role = "aut"),
    person("Thomas Lin", "Pedersen", role = "aut"),
    person("Kohske", "Takahashi", role = "aut"),
    person("Claus", "Wilke", role = "aut"),
    person("Kara", "Woo", role = "aut"),
    person("Hiroaki", "Yutani", role = "aut"),
    person("RStudio", role = c("cph"))
    )
Depends: R (>= 3.2)
Imports: digest, grDevices, grid, gtable (>= 0.1.1), lazyeval, MASS,
        mgcv, reshape2, rlang (>= 0.3.0), scales (>= 0.5.0), stats,
        tibble, viridisLite, withr (>= 2.0.0)
Suggests: covr, dplyr, ggplot2movies, hexbin, Hmisc, knitr, lattice,
        mapproj, maps, maptools, multcomp, munsell, nlme, profvis,
        quantreg, rgeos, rmarkdown, rpart, sf (>= 0.7-3), svglite (>=
        1.2.0.9001), testthat (>= 0.11.0), vdiffr (>= 0.3.0)
Enhances: sp
License: GPL-2 | file LICENSE
```

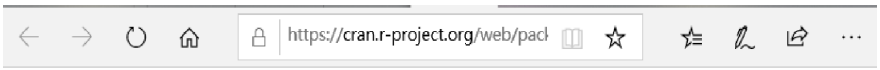
```
URL: http://ggplot2.tidyverse.org, https://github.com/tidyverse/ggplot2
BugReports: https://github.com/tidyverse/ggplot2/issues
LazyData: true
Collate: 'ggproto.r' ...
VignetteBuilder: knitr
RoxygenNote: 6.1.1
Encoding: UTF-8
NeedsCompilation: no
Packaged: 2019-08-09 20:11:46 UTC; thomas
Author: Hadley Wickham [aut, cre],
  Winston Chang [aut],
  Lionel Henry [aut],
  Thomas Lin Pedersen [aut],
  Kohske Takahashi [aut],
  Claus Wilke [aut],
  Kara Woo [aut],
  Hiroaki Yutani [aut],
  RStudio [cph]
Maintainer: Hadley Wickham <hadley@rstudio.com>
Repository: CRAN
Date/Publication: 2019-08-10 22:30:13 UTC
Built: R 3.6.1; ; 2019-10-26 06:08:11 UTC; windows
```

## Title and description: What does your package do?

- The title and description fields describe what the package does. They differ only in length:
  - `Title` is a one line description of the package, and is often shown in package listing.
    - It should be plain text (no markup), capitalised like a title, and NOT end in a period.
    - Keep it short: listings will often truncate the title to 65 characters.
  - `Description` is more detailed than the title.
    - You can use multiple sentences but you are limited to one paragraph.
    - If your description spans multiple lines (and it should!), each line must be no more than 80 characters wide. Indent subsequent lines with 4 spaces.

The `Title` and `Description` for `BeSS` are:

```
Title: Best Subset Selection in Linear, Logistic and CoxPH Models

Description: An implementation of best subset selection in generalized linear model
    and Cox proportional hazard model via the primal dual active set algorithm
    proposed by Wen, C., Zhang, A., Quan, S. and Wang, X. (2017) <arXiv:1709.06254>.
    The algorithm formulates coefficient parameters and residuals as primal and dual
    variables and utilizes efficient active set selection strategies based on the
    complementarity of the primal and dual variables.
```




## BeSS: Best Subset Selection in Linear, Logistic and CoxPH Models

An implementation of best subset selection in generalized linear model and Cox proportional hazard model via the primal dual active set algorithm proposed by Wen, C., Zhang, A., Quan, S. and Wang, X. (2017) <arXiv:1709.06254>. The algorithm formulates coefficient parameters and residuals as primal and dual variables and utilizes efficient active set selection strategies based on the complementarity of the primal and dual variables.

| | |
|---|---|
| Version: | 1.0.6 |
| Depends: | R (≥ 3.2.0) |
| Imports: | Rcpp (≥ 0.12.8), Matrix (≥ 1.2-6), glmnet, survival |
| LinkingTo: | Rcpp, RcppEigen |
| Published: | 2019-02-21 |
| Author: | Canhong Wen, Aijun Zhang, Shijie Quan, Xueqin Wang |
| Maintainer: | Canhong Wen <wencanhong at gmail.com> |
| License: | GPL-3 |
| NeedsCompilation: | yes |
| CRAN checks: | BeSS results |

### Downloads:

| | |
|---|---|
| Reference manual: | BeSS.pdf |
| Package source: | BeSS_1.0.6.tar.gz |
| Windows binaries: | r-devel: BeSS_1.0.6.zip, r-devel-gcc8: BeSS_1.0.6.zip, r-release: BeSS_1.0.6.zip, r-oldrel: BeSS_1.0.6.zip |
| OS X binaries: | r-release: BeSS_1.0.6.tgz, r-oldrel: BeSS_1.0.6.tgz |
| Old sources: | BeSS archive |

### Linking:

Please use the canonical form https://CRAN.R-project.org/package=BeSS to link to this page.

## Author: who are you?

- Use separate `Maintainer` and `Author` fields.

```
Author: Canhong Wen, Aijun Zhang, Shijie Quan, Xueqin Wang
Maintainer: Canhong Wen <wencanhong@gmail.com>
```

- Or more professional

```
Authors@R: c(
    person("Hadley", "Wickham", email = "hadley@rstudio.com", role = "cre"),
    person("Winston", "Chang", email = "winston@rstudio.com", role = "aut"))
```

```
+ A three letter code specifying the `role`. There are four important roles:
    + `cre`: the creator or maintainer, the person you should bother if you have problems.
    + `aut`: authors, those who have made significant contributions to the package.
    + `ctb`: contributors, those who have made smaller contributions, like patches.
    + `cph`: copyright holder. This is used if the copyright is held by someone other
             than the author, typically a company (i.e. the author's employer).
```

## Dependencies: What does your package need?

It's the job of the `DESCRIPTION` to list the packages that your package needs to work. R has a rich set of ways of describing potential dependencies.

```
Imports:
    Rcpp,
    Matrix
```

```
Suggests:
    Rcpp,
    Matrix
```

## Versioning

If you need a specific version of a package, specify it in parentheses after the package name:

```
Imports:
    Rcpp(>= 0.12.8),
    Matrix(>= 1.2-6)
Suggests:
    Rcpp(>= 0.12.8),
    Matrix(>= 1.2-6)
```

- Versioning is most important when you release your package.
- Generally, it's always better to specify the version and to be conservative about which version to require. Unless you know otherwise, always require a version greater than or equal to the version you're currently using.

## Other dependencies

There are three other fields that allow you to express more specialised dependencies:

- `Depends`: You can also use Depends to require a specific version of R, e.g. Depends: `R (>= 3.0.1)`.

- `LinkingTo`: packages listed here rely on C or C++ code in another package.
- `Enhances`: packages listed here are "enhanced" by your package. Typically, this means you provide methods for classes defined in another package (a sort of reverse `Suggests`).

## License: Who can use your package?

- **MIT**. This is a simple and permissive license. It lets people use and freely distribute your code subject to only one restriction: the license must always be distributed with the code.
- **GPL-2 or GPL-3**. These are "copy-left" licenses. This means that anyone who distributes your code in a bundle must license the whole bundle in a GPL-compatible way. Additionally, anyone who distributes modified versions of your code (derivative works) must also make the source code available. GPL-3 is a little stricter than GPL-2, closing some older loopholes.
- **CC0**. It relinquishes all your rights on the code and data so that it can be freely used by anyone for any purpose. This is sometimes called putting it in the public domain, a term which is neither well-defined nor meaningful in all countries.

## Version of your package

- A released version number consists of three numbers, <major>.<minor>.<patch>.
- For version number 1.9.2, 1 is the major number, 9 is the minor number, and 2 is the patch number.

## Part III

*Object documentation*

## Object documentation

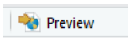
- Documentation is one of the most important aspects of a **good package**.
- R provides a standard way of documenting the objects in a package: you write `.Rd` files in the `man/` directory.
  - By hand
  - `roxygen2`
- These files use a custom syntax, loosely based on LaTeX, and are rendered to HTML, plain text and pdf for viewing.

## Advanrages with `roxygen2`

- Code and documentation are intermingled so that when you modify your code, you're reminded to also update your documentation.
- `Roxygen2` dynamically inspects the objects that it documents, so you can skip some boilerplate that you'd otherwise need to write by hand.
- It abstracts over the differences in documenting different types of objects, so you need to learn fewer details.

## The documentation workflow

1. Add roxygen comments to your `.R` files.
2. Run `devtools::document()` or press `Ctrl/Cmd + Shift + D` to convert roxygen comments to `.Rd` files. (`devtools::document()` calls `roxygen2::roxygenise()` to do the hard work.)




3. Preview documentation with `?` or
4. Rinse and repeat until the documentation looks the way you want.

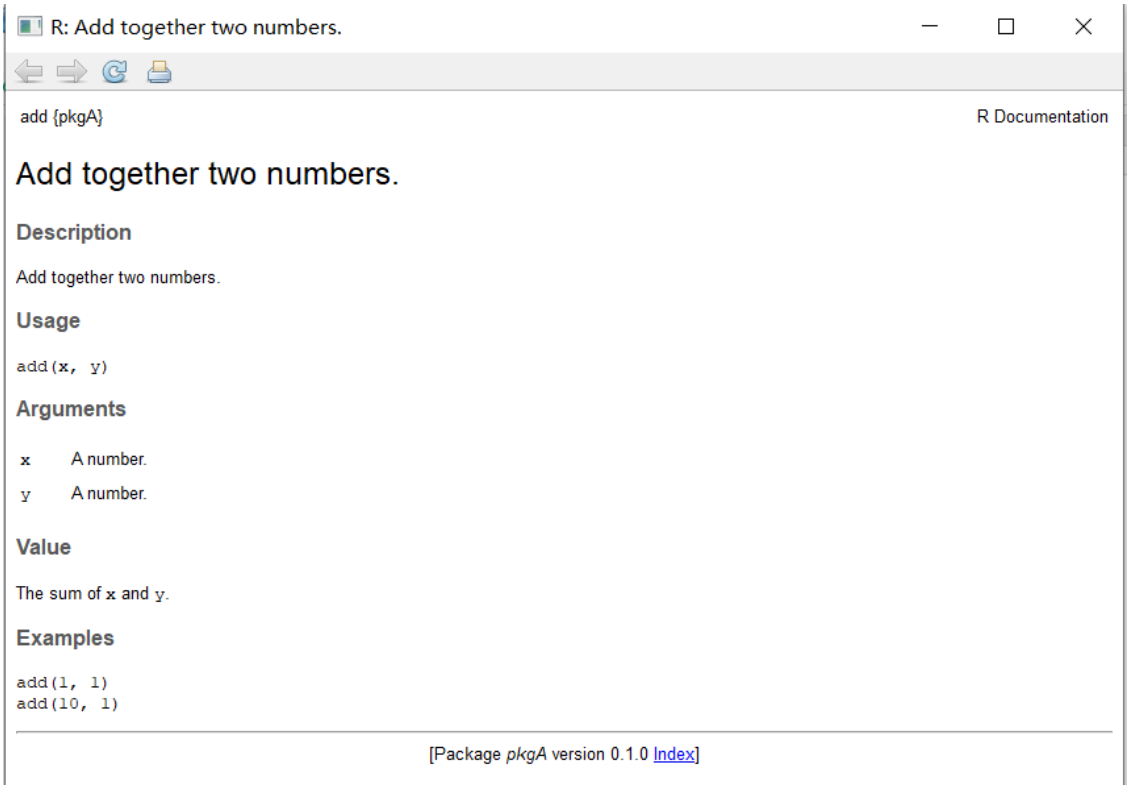
## Simple example

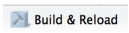
The R code in `.R` file

```
#' Add together two numbers.
#'
#' @param x A number.
#' @param y A number.
#' @return The sum of \code{x} and \code{y}.
#' @examples
#' add(1, 1)
#' add(10, 1)
add <- function(x, y) {
  x + y
}
```

The `.Rd` file looks like:

```
% Generated by roxygen2: do not edit by hand
% Please edit documentation in R/add.R
\name{add}
\alias{add}
\title{Add together two numbers.}
\usage{
add(x, y)
}
\arguments{
\item{x}{A number.}

\item{y}{A number.}
}
\value{
The sum of \code{x} and \code{y}.
}
\description{
Add together two numbers.
}
\examples{
add(1, 1)
add(10, 1)
}
```




## Alternative documentation workflow

The first documentation workflow is very fast, but it has one limitation: the preview documentation pages will not show any links between pages. If you'd like to also see links, use this workflow:

1. Add roxygen comments to your .R files.
2. Click
   in the build pane or press `Ctrl/Cmd + Shift + B`. This completely rebuilds the package, including updating all the documentation, installs it in your regular library, then restarts R and reloads your package. This is slow but thorough.
3. Preview documentation with `?` or `help()`.
4. Rinse and repeat until the documentation looks the way you want.

## Roxygen comments

- Roxygen comments start with #' and come before a function.
- All the roxygen lines preceding a function are called a **block**.
- Blocks are broken up into tags, which look like `@tagName details`. The content of a tag extends from the end of the tag name to the start of the next tag (or the end of the block).
- Each block includes some text before the first tag. This is called the **introduction**, and is parsed specially:
  - The first sentence becomes the **title** of the documentation. It should fit on one line, be written in sentence case, but not end in a full stop.

  - The second paragraph is the **description**: this comes first in the documentation and should briefly describe what the function does.
  - The third and subsequent paragraphs go into the **details**: this is a (often long) section that is shown after the argument description and should go into detail about how the function works.
- All objects must have a title and description. Details are optional.

### Example: Sum

```
#' Sum of vector elements.
#'
#' \code{sum} returns the sum of all the values present in its arguments.
#'
#' This is a generic function: methods can be defined for it directly or via the
#' \code{\link{Summary}} group generic. For this to work properly, the arguments
#' \code{...} should be unnamed, and dispatch is on the first argument.
sum <- function(..., na.rm = TRUE) {}
```

There are two tags that make it easier for people to navigate between help files:

- **@seealso**: allows you to point to other useful resources, either on the web, `\url{http://www.r-project.org}`, in your package `\code{\link{functioname}}`, or another package `\code{\link[packagename]{functioname}}`.
- **@family**: If you have a family of related functions where every function should link to every other function in the family, use @family. The value of @family should be plural.
- For sum, these components might look like:

```
#' @family aggregate functions
#' @seealso \code{\link{prod}} for products, \code{\link{cumsum}} for cumulative
#'   sums, and \code{\link{colSums}}/\code{\link{rowSums}} marginal sums over
#'   high-dimensional arrays.
```

### Documenting functions

Most functions have three tags: @param, @examples and @return.

- **@param** name description describes the function's inputs or parameters.
  - The description should provide a succinct summary of the type of the parameter (e.g., string, numeric vector) and, if not obvious from the name, what the parameter does.
  - The description should start with a capital letter and end with a full stop.
  - You can document multiple arguments in one place by separating the names with commas (no spaces).
- **@examples** provides executable R code showing how to use the function in practice.
  - Example code must work without errors as it is run automatically as part of `R CMD check`.
  - For the purpose of illustration, it's often useful to include code that causes an error. `\dontrun{}` allows you to include code in the example that is not run.
- **@return** description describes the output from the function.

```
#' Sum of vector elements.
#'
#' \code{sum} returns the sum of all the values present in its arguments.
#'
#' This is a generic function: methods can be defined for it directly
#' or via the \code{\link{Summary}} group generic. For this to work properly,
#' the arguments \code{...} should be unnamed, and dispatch is on the
#' first argument.
#'
#' @param ... Numeric, complex, or logical vectors.
#' @param na.rm A logical scalar. Should missing values (including NaN)
#'   be removed?
#' @return If all inputs are integer and logical, then the output
#'   will be an integer. If integer overflow
#'   \url{http://en.wikipedia.org/wiki/Integer_overflow} occurs, the output
#'   will be NA with a warning. Otherwise it will be a length-one numeric or
#'   complex vector.
#'
#'   Zero-length vectors have sum 0 by definition. See
#'   \url{http://en.wikipedia.org/wiki/Empty_sum} for more details.
#' @examples
#' sum(1:10)
#' sum(1:5, 6:10)
#' sum(F, F, F, T, T)
#' sum(.Machine$integer.max, 1L)
#'
#' \dontrun{
#' sum("a")
#' }
sum <- function(..., na.rm = TRUE) {}
```

## Documenting packages

- You can use roxygen to provide a help page for your package as a whole.
- Put this documentation in a file called `<package-name>.R`.

```
#' foo: A package for computating the notorious bar statistic.
#'
#' The foo package provides three categories of important functions:
#' foo, bar and baz.
#'
#' @section Foo functions:
#' The foo functions ...
#'
#' @docType package
#' @name foo
NULL
```

```
## NULL
```

Help on topic 'flights' was found in the following packages:

Flights data
(in package nycflights13 in library C:/Users/Canhong Wen/Documents/R/win-library/3.6)
Flights data
(in package pkgA in library C:/Users/Canhong Wen/Documents/R/win-library/3.6)

## Part IV

*Vignetters*

## Vignettes: long-form documentation

- A vignette is a long-form guide to your package.
- Function documentation is great if you know the name of the function you need, but it's useless otherwise.
- A vignette is like a book chapter or an academic paper: it can describe the problem that your package is designed to solve, and then show the reader how to solve it.
- `browseVignettes("packagename")`
- Using `rmarkdown` to write your own vignette.

## Example

```
---
title: "Vignette Title"
author: "Vignette Author"
date: "`r Sys.Date()`"
output: rmarkdown::html_vignette
vignette: >
  %\VignetteIndexEntry{Vignette Title}
  %\VignetteEngine{knitr::rmarkdown}
  \usepackage[utf8]{inputenc}
---
```

## Part V

*Namespaces*

## Motivation

- namespaces provide "spaces" for "names".

```
?flights
```

Help on topic 'summarize' was found in the following packages:

Summarise a data frame.
(in package plyr in library C:/Users/Canhong Wen/Documents/R/win-library/3.6)
Summarize Scalars or Matrices by Cross-Classification
(in package Hmisc in library C:/Users/Canhong Wen/Documents/R/win-library/3.6)
Reduce multiple values down to a single value
(in package dplyr in library C:/Users/Canhong Wen/Documents/R/win-library/3.6)

```
> library(Hmisc)
載入程輯包: 'Hmisc'
The following objects are masked from 'package:base':
    format.pval, units
> library(plyr)
載入程輯包: 'plyr'
The following objects are masked from 'package:Hmisc':
    is.discrete, summarize
```

### `summarize()`

- both `plyr` and `Hmisc` provide a `summarize()` function.
  - If you load `plyr`, then `Hmisc`, `summarize()` will refer to the `Hmisc` version.
  - But if you load the packages in the opposite order, `summarize()` will refer to the `plyr` version.
- you can explicitly refer to specific functions: `Hmisc::summarize()` and `plyr::summarize()`.

## The imports and the exports

Namespaces make your packages self-contained in two ways: the **imports** and the **exports**.

- The **imports** defines how a function in one package finds a function in another.
- The **exports** helps you avoid conflicts with other packages by specifying which functions are available outside of your package.

## An illustrative example

```
nrow
```

```
## function (x)
## dim(x)[1L]
## <bytecode: 0x000000000998b880>
```

```
## <environment: namespace:base>
```

```
dim(mtcars)
```

```
## [1] 32 11
```

```
dim <- function(x) c(1, 1)
dim(mtcars)
```

```
## [1] 1 1
```

```
nrow(mtcars)
```

```
## [1] 32
```

## Namespaces

- Original file

```
exportPattern("^[[:alpha:]]+")
```

- After we add the flights data, and generate the `NAMESPACE` file with `roxygen2`

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

importFrom(tibble,tibble)
```

```
useDynLib(BeSS, .registration = TRUE)
importFrom(Rcpp, evalCpp)
importFrom("survival", "coxph")
importFrom("survival", "Surv")
importFrom("glmnet", "glmnet")
importFrom("Matrix", "Matrix")
importFrom("stats", "binomial", "lm","deviance", "logLik")
importFrom("graphics","abline","axis","box","grid","layout","lines","mtext","par",
           "plot","plot.new","plot.window","text","title")
importFrom("stats","glm","rbinom","rnorm","runif","model.matrix")


export(bess, bess.one, gen.data, aic, bic, gic)

S3method(plot,bess)
S3method(coef,bess)
S3method(print,bess)
S3method(summary,bess)
S3method(predict,bess)
S3method(logLik,bess)
S3method(deviance,bess)
S3method(coef,bess.one)
S3method(print,bess.one)
S3method(summary,bess.one)
S3method(predict,bess.one)
S3method(logLik,bess.one)
S3method(deviance,bess.one)
```

- You can see that the NAMESPACE file looks a bit like R code.
- Each line contains a directive: `S3method()`, `export()`, `exportClasses()`, and so on.
- In total, there are eight namespace directives. Four describe exports:
  - `export()`: export functions (including S3 and S4 generics).
  - `exportPattern()`: export all functions that match a pattern.
  - `exportClasses()`, `exportMethods()`: export S4 classes and methods.
  - `S3method()`: export S3 methods.
- And four describe imports:
  - `import()`: import all functions from a package.
  - `importFrom()`: import selected functions (including S4 generics).
  - `importClassesFrom()`, `importMethodsFrom()`: import S4 classes and methods.
  - `useDynLib()`: import a function from C. This is described in more detail in compiled code.

# Part VI

*Data*

## Data

There are three main ways to include data in your package, depending on what you want to do with it and who should be able to use it:

- If you want to store binary data and make it available to the user, put it in `data/`. This is the best place to put example datasets.
- If you want to store parsed data, but not make it available to the user, put it in `R/sysdata.rda`. This is the best place to put data that your functions need.
- If you want to store raw data, put it in `inst/extdata`.

## Exported data

- The most common location for package data is `data/`.
- Each file in this directory should be a `.RData` file created by `save()` containing a single object (with the same name as the file).
- To load the data, just type `data(YourDataName)`
- For larger datasets, you may want to experiment with the compression setting. The default is `bzip2`.
- For more details, see `?data`.

```
d <- data(package = "pkgA")

## names of data sets in the package
d$results[, "Item"]
```

```
## [1] "flights"  "prostate"
```

- If the `DESCRIPTION` contains `LazyData: true`, then datasets will be lazily loaded. This means that they won't occupy any memory until you use them.

```
pryr::mem_used()
```

```
## Registered S3 method overwritten by 'pryr':
##   method      from
##   print.bytes Rcpp
```

```
## 39.5 MB
```

```
library(pkgA)

pryr::mem_used()
```

```
## 42.9 MB
```

```
data("flights")
invisible(flights)
pryr::mem_used()
```

```
## 83.6 MB
```

## Documenting datasets

```
#' Flights data
#'
#' On-time data for all flights that departed NYC (i.e. JFK, LGA or EWR) in 2013.
#'
#' @source RITA, Bureau of transportation statistics,
#'  <https://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236>
#' @format Data frame with columns
#' \describe{
#' \item{year, month, day}{Date of departure.}
#' \item{dep_time, arr_time}{Actual departure and arrival times (format HHMM or HMM)}
#' ...
#' }
"flights"

#' @importFrom tibble tibble
NULL
```

There are two additional tags that are important for documenting datasets:

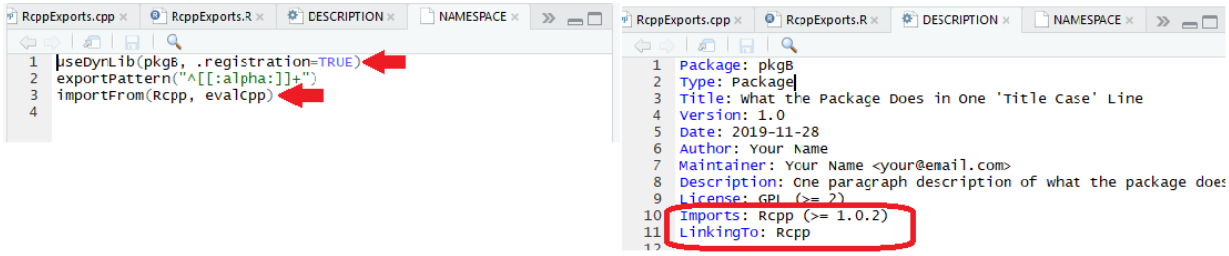
- @format gives an overview of the dataset. For data frames, you should include a definition list that describes each variable.
- @source provides details of where you got the data, often a `\url{}`.

# Part VII

*Compiled code*

## Compiled code

- Recommend starting with C++ and the Rcpp package
- Reference:
  1. Hadley Wickham's Advanced R: High performance functions with Rcpp


2. Dirk Eddelbuettel's Seamless R and C++ Integration with Rcpp
3. Rcpp documentation: Rcpp Version 1.0.3.1 Documentation

# C++

## Workflow

1. Create a new C++ file:
2. Generate the necessary modifications to your `NAMESPACE` by documenting them with `Ctrl/Cmd + Shift + D`.
3. Click `Build & Reload` in the build pane, or press `Ctrl/Cmd + Shift + B`.
4. Run the R function from the console to check that it works.

## A simple example

```
#include <Rcpp.h>
using namespace Rcpp;

// This is a simple example of exporting a C++ function to R. You can
// source this function into an R session using the Rcpp::sourceCpp
// function (or via the Source button on the editor toolbar). Learn
// more about Rcpp at:
//
//   http://www.rcpp.org/
//   http://adv-r.had.co.nz/Rcpp.html
//   http://gallery.rcpp.org/
//
```

```
// [[Rcpp::export]]
NumericVector timesTwo(NumericVector x) {
  return x * 2;
}


// You can include R code blocks in C++ files processed with sourceCpp
// (useful for testing and development). The R code will be automatically
// run after the compilation.
//

/*** R
timesTwo(42)
*/
```

## A simple example

- The two most important parts are the header `#include`, and the special attribute `// [[Rcpp::export]]`.
- Each exported C++ function automatically gets a wrapper function (it will be located in `R/RcppExports.R`). For example, the R `timesTwo()` function looks like:

```
timesTwo <- function(x) {
    .Call(`_pkgB_timesTwo`, x)
}


timesTwo(1)
timesTwo(10)
```

## Documentation

You can use roxygen2 to document this like a regular R function. But instead of using `#'` for comments use `//'`

```
//' Multiply a number by two
//'
//' @param x A single interger.
//' @export
// [[Rcpp::export]]
NumericVector timesTwo(NumericVector x) {
  return x * 2;
}
```

That generates roxygen comments in `R/RcppExports.R`:

```
#' Multiply a number by two
#'
#' @param x A single interger.
#' @export
timesTwo <- function(x) {
    .Call(`_pkgB_timesTwo`, x)
}
```

And genenrate `.Rd` like this

```
% Generated by roxygen2: do not edit by hand
% Please edit documentation in R/RcppExports.R
\name{timesTwo}
\alias{timesTwo}
\title{Multiply a number by two}
\usage{
timesTwo(x)
}
\arguments{
\item{x}{A single interger.}
}
\description{
Multiply a number by two
}
```

## Importing C++ code

To use C++ code from another package:

- In `DESCRIPTION`, add `LinkingTo: otherPackage`. It adds otherPackage/include to the include path, allowing you to dynamically "link to??? other code via the headers.
- In the `C++` file, add:

  ```
  #include <otherPackage.h>
  ```

- `C++` functions from otherPackage will be included in the `otherPackage` namespace. Use `otherPackage::foo()` to access functions, or make them available globally with using `namespace otherPackage`.

## C

- If you're writing new compiled code, it's almost always better to use Rcpp. It's less work, more consistent, better documented, and it has better tools.
- However, there are some reasons to choose C:
  - You're working with an older package that already uses the C API.
  - You're binding to an existing C library.

## Workflow

1. Modify the C code.
2. Build and reload the package with Ctrl/Cmd + Shift + B
3. Experiment at the console.

The first time you add `@useDynLib`, you'll also need to run `devtools::document() (Ctrl/Cmd + Shift + D)` and reload the package.

### Getting started with `.C()`

To use it, you first write a void C function, using in-place modification of function parameters to return values:

```
void add_(double* x, double* y, double* out) {
  out[0] = x[0] + y[0];
}
```

Then create an R wrapper:

```
#' @useDynLib mypackage add_
add <- function(x, y) {
  .C(add_, x, y, numeric(1))[[3]]
}
```

(Here we extract the 3rd element of the result because that corresponds to the out parameter.)

- `.C()` automatically converts back and forth between R vectors and their C equivalents:

| R type | C type |
|---|---|
| logical | int* |
| integer | int* |
| double | double* |
| character | char* |
| raw | unsigned char* |

- `.C()` assumes your function doesn't know how to deal with missing values and will throw an error if any arguments contain an NA. If it can correctly handle missing values, set NAOK = TRUE in the call to `.C()`.

### Importing C code

Using C code from another package varies based on how the package is implemented:

- If it uses the system described above, all you need is `LinkingTo: otherPackage` in the `DESCRIPTION`, and `#include otherPackageAPI.h` in the C file.
- If it registers the functions, but doesn't provide a header file, you'll need to write the wrapper yourself. Since you're not using any header files from the package, use `Imports` and not `LinkingTo`. You also need to make sure the package is loaded. You can do this by importing any function with `@importFrom mypackage foo`, or by adding `requireNamespace("mypackage", quietly = TRUE)` to `.onLoad()`.
- If it doesn't register the functions, you can't use them. You'll have to ask the maintainer nicely or even provide a pull request.

### Other languages

It is possible to connect R to other languages, but the interfaces are not as nice as the one for C++:

- **Fortran**: It's possible to call Fortran subroutines directly with `.Fortran()`, or via C or C++ with `.Call()`. See `?.Fortran`
- **Java**: The `rJava` package makes it possible to call Java code from within R.