# Building R Packages

Kylie Ariel Bemis

11/7/2017

## References for building R packages

*R packages* by Hadley Wickam

`- Book freely available at http://r-pkgs.had.co.nz`

*Writing R Extentions* by the R Core Team

`- https://cran.r-project.org/doc/manuals/r-release/R-exts.html`

## Why build an R package?

Even if you are not planning on distributing your code, building an R package has many advantages over `source()`-ing your R scripts.

- Easily manage and organize your code
- Portably re-use your code
- Documentation for your code
- Make dependencies on other packages explicit
- Distribute your code! (internally or externally)

## Why build an R package? (cont'd)

Although many of the steps we will discuss today are R-specific, most of what you will learn can be applied to managing any type of projects that involve code. These include:

- Organizing your code for reproducibility
- Writing code that obeys user expectations
- Writing documentation and unit tests

## Getting started

We will use a few packages that help in the task of creating and developing R packages.

R packages can be authored without any of these tools, but they make life a little bit easier.

```r
install.packages(c("devtools", "roxygen2", "testthat"))
```

- `devtools` helps with all aspects of building and installing packages
- `roxygen2` streamlines writing documentation for your code
- `testthat` streamlines writing unit tests for your code

## Creating a template

To get started, devtools and base R provide us with two ways of creating a template for a new package.

```r
library(devtools)
devtools::create("pkgtemplate1")
```

```r
package.skeleton("pkgtemplate2")
```

Using devtools::create() is slightly more minimalist, but it still creates some files that are helpful but not strictly necessary.

Explore the directories that are created by running of both functions.

## Package structure

An R package is really just a directory with certain required files and subdirectories. The directory layout of an R package is:

- `PACKAGE_NAME/`
  - `DESCRIPTION` : A structured description of the package
  - `NAMESPACE` : Imports package dependencies and exports your functions and classes
  - `R/` : All your R code lives here
  - `man/` : Documentation of exported R functions and classes
  - `src/` : Compiled code (C, C++, Fortran, etc.) lives here
  - `inst/` : Miscellaneous files to move into the package directory when installed
  - `data/` : `.RData` or `.rda` data files used by the package
  - `tests/` : Unit tests for the package's functions and classes
  - `vignettes/` : Vignettes illustrating the package's use
  - `NEWS` : Changelog for the package

Many of these directories are optional.

## Package structure (cont'd)

- `PACKAGE_NAME/`
  - `DESCRIPTION` : A structured description of the package
  - `NAMESPACE` : Imports package dependencies and exports your functions and classes
  - `R/` : All your R code lives here
  - `man/` : Documentation of exported R functions and classes

R will build and install a package with only the DESCRIPTION file and `R/` subdirectory. However, the minimal requirements for a proper package also include a NAMESPACE file and ".Rd" documentation in the `man/` subdirectory for all exported functions and classes.

All of these can be created by hand. However, some tools exist that can generate some of these automatically.

## Tools for automation

- devtools
  - Build, check, and install packages from within R
  - Offers convenience functions for adding common development infrastructure to a package
    - devtools::document() for roxygen2::roxygenize()
    - devtools::use_testthat() for adding testthat infrastructure template
    - devtools::use_vignette() for adding a R Markdown vignette template.
    - devtools::use_rcpp() for Rcpp infrastructure template, etc.
- roxygen2
  - Inspired by doxygen documentation generator for C++, Java, Python, etc.
  - Automatically generate R documentation from structured code comments
  - Automatically generate NAMESPACE file from structured code comments
  - Will discuss more when discussing man/ and NAMESPACE
- testthat
  - R package that provides convenience functions for writing unit tests
  - Other such packages exist for the same purpose, such as RUnit
  - Will discuss more when discussing tests/

## How do packages work anyway?

A single package may exist in 5 states:

- Source package
- Bundled package
- Binary package
- Installed package
- In-memory package

## Source package

- This is just a directory with the structure discussed previously (contains `DESCRIPTION`, `R/`, etc.)

## Bundled package

- This is a source package that has been bundled and compressed into a single file with extension `.tar.gz` (often called a tarball)
- All files and directories in the source package are reduced to a single file using the *nix `tar` utility, and then compressed using `gzip`
- Often refered to as “source packages” because they are a compressed version of a source package
- Built with `R CMD build` or `devtools::build()`

## Binary package

- R code in `R/` has been parsed, evaluated, and the output saved as `.rds` files for fast loading
- Compiled code in `src/` has been compiled to binary executables for their respectibe operating systems and architectures
- Documentation and vignettes have been converted to HTML and/or PDF
- Contents of `inst/` are moved to top-level package directory
- Are Mac- or Windows- specific (and do not exist on Linux)
- Built with `R CMD INSTALL --build` or `devtools::build(binary=TRUE)`

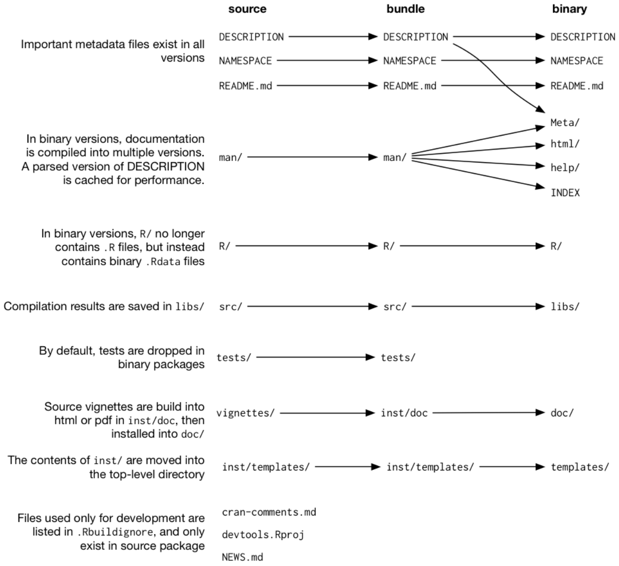
source
bundle
binary
Important metadata files exist in all versions
DESCRIPTION
NAMESPACE
README.md
DESCRIPTION
NAMESPACE
README.md
DESCRIPTION
NAMESPACE
README.md
Meta/
In binary versions, documentation is compiled into multiple versions. A parsed version of DESCRIPTION is cached for performance.
man/
man/
html/
help/
INDEX
In binary versions, R/ no longer contains .R files, but instead contains binary .Rdata files
R/
R/
R/
Compilation results are saved in libs/
src/
src/
libs/
By default, tests are dropped in binary packages
tests/
tests/
Source vignettes are build into html or pdf in inst/doc, then installed into doc/
vignettes/
inst/doc
doc/
The contents of inst/ are moved into the top-level directory
inst/templates/
inst/templates/
templates/
Files used only for development are listed in .Rbuildignore, and only exist in source package
cran-comments.md
devtools.Rproj
NEWS.md

Figure 1: Bundled vs binary packages

## Installed package

- A decompressed version of a binary package that has been saved to a package library
  - Note that despite the confusingly-named `library()` function, a “library” in R is not the same thing as a package; rather it is a collection of installed packages
  - You typically have 2 libraries at any given time: one for all the default packages that come with R, and one for all of the packages you have manually downloaded and installed
- Install with `R CMD INSTALL` or `devtools::install()`

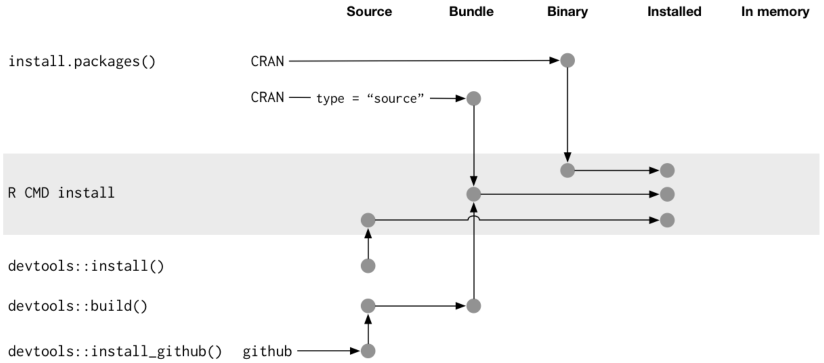

Figure 2: Installed packages

## In-memory packages

- An installed package that has been loaded into memory during an R session
- Note that a package may be loaded into memory but *not* be attached to the search path
- Packages are automatically loaded whenever you use a function from that package
  - E.g. you can use `dplyr::summarise()` without calling `library(dplyr)` first, and the `dplyr` package will automatically be loaded into memory, but *not* attached to the search path
  - What `library(dplyr)` actually does is both load the package into memory *and* attach its namespace to the search path

## Workflow for packages

In order to actually go from a source package to loading a package in R, there are 3 required steps (and 2 optional) steps.

- Build a bundled version of the package with `R CMD build` or `devtools::build()`
  - Optionally, check the package with `R CMD check` or `devtools::check()`
- Install the bundled package with `R CMD INSTALL` or `devtools::install()`
  - Optionally, build a binary version of the package with `R CMD INSTALL --build` or `devtools::build(binary=TRUE)`
  - Note that binary packages are actually built by compressing the installed version of a package
- In a new R session, load the package and attach it to the search path with `library()`

## Workflow for packages

In practice, `devtools` automates many of the steps of building and installing packages.

- `devtools::build()` builds bundled *or* binary packages
- `devtools::check()` build *and* checks packages
- `devtools::install()` builds *and* installed packags
- `devtools::load_all()` builds, installs, and loads a package
  - Note that `load_all()` is intended for rapid iteration and testing new code during development and actually *simulates* some of the steps rather than actually performing them, so it should not be used in place of actually building and installing a package

## Example package: `regr`

In order to learn about package development, we will use a minimal package that implements linear regression, similar to `lm()`.

The `regr` package implements the `reg()` function for linear regression. The bundled package is provided for you as `regr_0.0.1.tar.gz`.

## Package metadata (`DESCRIPTION`)

The DESCRIPTION file lists various attributes of the package, including:

```
Package: The name of the package
Title: A more descriptive name of the package
Version: ?.?.?
Author: Who authored the package
Maintainer: Who currently maintains the package
Description: A longer-form description of the package functional
Depends: A list of packages that must be loaded and attached
Imports: A list of packages that must be loaded but are not atta
Suggests: A list of packages used by tests, vignettes, etc.,
          but are not needed for the package to function
License: A legal license (e.g., GPL, MIT, etc.)
```

## The description file for the regr package

```
Package: regr
Title: Linear regression
Version: 0.0.1
Author: Kylie Bemis
Maintainer: Kylie Bemis <k.bemis@northeastern.edu>
Description: This package implements the reg function for linear
Depends: R (>= 3.4.2), stats, modelr
Suggests: ggplot2, roxygen2, testthat, knitr
License: Artistic-2.0
LazyData: true
RoxygenNote: 6.0.1
VignetteBuilder: knitr
```

## Code (`R/`)

This subdirectory contains all of your R code in `.R` scripts. You may arrange your code into as many or as few scripts as you like, but it is a good idea to be organized.

A few things to remember about code in `R/`:

- All the R code will be evaluated when the package is built, NOT when it is loaded into R by a user
- All the R code will be evaluated in an environment that will become the namespace of the package, NOT in the global environment
- Any R objects (including functions) you assign in this R code will *not* be visible to the user or other packages unless it is *exported* in the `NAMESPACE`
- It should mostly be functions, methods, and class definitions

## Code (`R/`) (cont'd)

- You should not use code that creates side-effects outside of functions
  - E.g., write a plot function, don't try to plot something when the package starts up
- You should not change the user's global environment, working directory, or options, or read or write files or communicate over the internet without the user's permission (i.e., via the user calling a function that explicitly does this)
- All code will be evaluated in the order they appear (R scripts will be evaluated in alphabetical order)
  - For functions, this makes no difference, because arguments are evaluated lazily, and the body is not evaluated until the function is called
  - For S4 classes and methods, a class must be be defined before you can define methods for it, so the order *does* matter
  - Use `Collate:` field in `DESCRIPTION` file to specify an order in which to evaluate the R scripts in `R/`

## Documentation (`man/`)

This directory includes all of the documentation for the package and R objects (including functions, methods, class definitions, and datasets).

It is written in `.Rd` files which are based on `LaTeX`.

These files can be authored by hand, but we will use the package `roxygen2` which allows us to keep our documentation with our code by using code comments. Similar to `doxygen`, `roxygen2` generates the documentation for us from structured code comments that we put near the function we want to document.

## Documenting a function

```
#' Linear regression
#'
#' \code{reg} fits a linear regression model.
#'
#' @param formula The model formula.
#' @param data The data to use.
#'
#' @return An object of class \code{reg}.
#'
#' @examples
#' library(datasets)
#' reg(mpg ~ disp, data=mtcars)
#'
#' @export
reg <- function(formula, data) {
  regFit(fm=as.formula(formula), data=data)
}
```

## Documenting methods

```
#' @describeIn reg Predict method for reg objects
#'
#' @param newdata New data to use for prediction.
#'
#' @method predict reg
#' @export
predict.reg <- function(object, newdata, ...) {
  Xnew <- model.matrix(object$formula, newdata)
  B <- as.matrix(object$coefficients)
  as.vector(Xnew %*% B)
}
```

## Documenting the package

```
#' regr: Linear regression
#'
#' The regr package implement basic linear regression
#'
#' @docType package
#' @name regr
#'
#' @import stats
#' @import modelr
NULL
```

## Generating documentation from roxygen

Call `roxygen2::roxygenize()` or `devtools::document()` to generate the `.Rd` files in the `man/` subdirectory from the code comments.

```r
roxygen2::roxygenize("regr")
```

```r
devtools::document("regr")
```

## Namespace (`NAMESPACE`)

The namespace uses directives such as `import()` and `export()` to specify any R objects to import from other packages, and which to export to the user.

- `import` makes all functions in a package available to your package
- `importFrom` selectively imports functions for use from another package
- `export` exports any R object such as functions to be available to the user and other packages
- `S3method` exports S3 methods
- `exportClasses` exports S4 classes
- `exportMethods` exports S4 methods

## The generated namespace for `regr`

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

S3method(predict,reg)
S3method(print,reg)
S3method(summary,reg)
export(reg)
import(modelr)
import(stats)
```

## Data (`data/`)

This data simply includes any data (saved as `.Rdata` or `.rda` using the `save()` or `saveRDS()` functions.

When the package is loaded, this data is not automatically loaded. Instead, they are loaded on-demand using `data()`.

Any datasets included in a package must also be documented. They can be documented using `roxygen2` similarly to functions.

## Installed files (`inst/`)

Any files in this directory are moved to the top-level directory when the package is installed.

This might be useful for:

- Data that is not in `.Rdata` or `.rda` format, such as CSV files
- Additional documentation
- Scripts for non-compiled langauges that the package uses like Python

## Compiled code (`src/`)

This is the directory for any C, C++, or FORTRAN compiled code that the package uses.

We will discuss this directory more next time when discussing Rcpp.

## Vignettes (`vignettes/`)

Vignettes are (typically) R Markdown files that are compiled when the package is built.

You can add a template for a vignette using:

```r
devtools::use_vignette(name="regr", pkg="regr")
```

The `regr` package has a vignette you can view via:

```r
vignette("regr")
```

## Unit tests (`tests/`)

Unit tests are used to test small, reproducible snippets of code to confirm they do what they say they do. You don't need to use any additional packages for this, but using a package such as `testthat` or `RUnit` make writing unit tests a bit easier.

You can create a template for writing unit tests with `testthat` by running:

```r
devtools::use_testthat(pkg="regr")
```

## The testthat.R file for regr

```r
library(testthat)
library(regr)

test_check("regr")
```

## The tests/test-regr.R file for regr

```r
context("regr - compare with lm")

test_that("reg output matches lm output", {

  library(ggplot2)

  fit_reg <- reg(hwy ~ displ, data=mpg)
  fit_lm <- lm(hwy ~ displ, data=mpg)
  expect_equivalent(coef(fit_reg), coef(fit_lm))

})
```

## Building a package

Building the package makes a bundled package out of a source package. This allows for easier distribution, and is the preferred input for checking and installing packages from the command line. You can build a package from the command line or from R.

`R CMD build` takes a source package as input.

```
R CMD build regr
```

`devtools::build()` takes a source package as input.

```r
devtools::build("regr")
```

## Checking a package

Checking a package looks for common code errors, checks the documentation, checks that the package can be loaded smoothly, and makes sure any examples or unit tests run without error. For CRAN or Bioconductor to accept a package, it must pass `R CMD check` without errors or warnings. Checking a package is best done on a bundled package, and can be done from the command line or in R.

`R CMD check` takes a bundled (or source) package as input.

```
R CMD check regr_0.0.1.tar.gz
```

`devtools::check()` takes a source package as input.

```
devtools::check("regr")
```

## Installing a package

To use the package in R, it must be installed. This can be done from the command line or in R. Installing a package is the only part of building/checking/installing packages that can be done from R without `devtools` (unless you use the `system()` function).

`R CMD INSTALL` takes a bundled (or source or binary) package as input.

```
R CMD INSTALL regr_0.0.1.tar.gz
```

`install.packages` takes a variety of argument types.

```r
install.packages("regr_0.0.1.tar.gz", repos=NULL, type="source")
```

`devtools::install()` takes a source package as input.

```r
devtools::install("regr")
```