Publishing packages on CRAN made simple

Stuart Lacy

20th September 2016

Aims

- To provide motivation to publish to CRAN
- Develop an awareness of package development best practices
- Demonstrate that publishing packages on CRAN is very achievable

Contents

Introduction to CRAN

Why CRAN?

- Why publish code?
  - Reproducibility
  - Extend the reach/impact of your work
  - Teaches you to generalise your ideas into algorithms
  - Develop software development skills
  - Contribute to open source
- Why CRAN in particular?
  - Forces you to write code to a higher standard of quality and organisation
  - Provides higher level of quality assurance for your users
  - Familiarises you with idiomatic R coding practices
  - Possibility of additional publication:
    - Journal of Statistical Software
    - The R Journal
  - Not that much extra work is required!

How to get started

- **Highly** recommend using RStudio
- Hadley's devtools package makes everything far simpler
- There is a very specific folder structure which must be adhered to, but RStudio/devtools takes care of this for you
- Using git for version control is recommended, either with github, another host, or just locally
- Note that unlike other programming languages which ship binary files, all your source code will be accessible on CRAN

Package Structure

First package

- Creating a package with RStudio[1] provides the following files:
  - R/ - Where source code (.R files) are stored
  - man/ - **Don't manually edit** - Documentation is saved here
  - DESCRIPTION - Package details, scraped by CRAN
  - NAMESPACE - **Don't manually edit** - Describes import/exports
  - *packagename*.Rproj - RStudio project file
  - .Rbuildignore
    - Text file containing list of files to not include when building, i.e. those which won't appear in the archive that is placed onto CRAN
    - By default contains just the .Rproj file, I also use it for TODO files and vignette resources

[1] File -> New Project -> New Directory -> Package

Example file structure of a completed source package

- Completed packages will contain additional content, for example:
    - data/ - Any data sets provided with the package (.csv, .txt, .Rda format)
    - tests/ - Unit tests
    - vignettes/ - All vignette source files
    - NEWS.md - Summarises version release history
    - README.md - Simple text file outlining the package and installation instructions
- I highly recommend looking at others' code when stuck, particularly someone like Hadley who is very meticulous with his coding style and maintains active Github repositories

Building packages

- Building a package transforms the raw source code into a distributable stand-alone package (in the form of a .zip archive)
- In the case of R, this typically is related to non-code aspects
    - Function documentation is generated from your docstring comments
    - Vignettes are compiled from Rmarkdown into HTML or PDF
    - The `NAMESPACE` file is updated
- Use `devtools::build()`, or RStudio offers buttons in that mysterious “Build” tab
- You don’t need to build the package until you’ve thoroughly tested it and decided it’s ready for distribution

Building packages - 2

- The resulting built packages can be viewed on CRAN, under ”Source Code” on a package’s page
- Can see our package rprev at https://cran.r-project.org/web/packages/rprev/index.html
- Package for estimating disease prevalence using Monte-Carlo simulation

Example file structure of a built package

- Additional files made when building package
  - build/ - **Can ignore**. Objects related to vignettes.
  - inst/ - Vignette documentation generated from source
  - man/ - Function documentation generated from docstrings
  - MD5 - **Can ignore**. Hash of each file contained within for validation.
- Never want to manually edit these folders/files

Documentation

Function documentation

- Simon Hickinbotham's talk gave examples of these https://github.com/franticspider/rpkgtalk
- Used to generate the help files viewed with `?function_name`
- Write documentation commments with `#'` above each function
- A program called roxygen2 converts these into standalone help documents
- To preview, run `devtools::document()`, this will place compiled documents in man folder. Can then view in RStudio as if package was installed

Docstring example

```
#' 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
}
```

Docstring example - 2

add {rvest} R Documentation

Add together two numbers

Description

Add together two numbers

Usage

```
add(x, y)
```

Arguments

`x`	A number
`y`	A number

Value

The sum of `x` and `y`

Examples

```
add(1, 1)
add(10, 1)
```

- Running `devtools::document()` makes the help page for this function viewable
- NB: Examples **must** be runnable!

Namespaces

Namespaces

- Each package has its own namespace, consisting of function names that it provides; can run into problems when multiple packages have the same named function
- Challenging concept to grasp, but need to have an awareness when developing packages
- **Never use library() or require() in a package!**
- What's the difference between `library(dplyr) filter(mydf, age<60)` and `dplyr::filter(mydf, age<60)`?
- Attaching `dplyr` using `library()` first means that I can access any of its other functions straight away, with the second method I'd still need to use the `::` syntax to access other functions
- Use `library()` and `require()` in analysis scripts, but **not** in package code

Namespace example

Package york

```
simulate <- function(x) {
    library(dplyr)
    # uses dplyr functions
    ...
    # return some value
}
```

User code

```
library(Hmisc)
library(york)

# dplyr is attached
# unbeknown to the
# user
vals <- simulate(x)

# user wants to use
# summarize from Hmisc
# but R will try and use
# dplyrs version
summarize(vals)
```

Namespaces overview

- Confusingly, we come into contact with namespaces in two distinct occasions when developing packages:
  1. The `DESCRIPTION` file
  2. At the point of use in the code itself, when we want to call functions from external packages

- What's the difference between these, and what should we do to ensure best namespace practices when developing packages?

Namespaces - DESCRIPTION

- In the `DESCRIPTION` file, we can list external dependencies as either Imports or Depends
- Depends:
  - attaches the package when yours is loaded, adding its entire namespace of functions to the current environment
  - As we've seen before, attaching external packages can lead to confusion for the user
  - **Only** use if your package is heavily dependent upon the external package and builds upon it
- Imports:
  - loads the package when yours is loaded, making it ready for use but doesn't add anything to the namespace
  - **Always** use Imports, unless you have a strong reason not to
- Suggests is also used for dependencies to build the package, but not run it, e.g. those required to compile vignette

Namespaces - Referencing external packages

- How do we handle referencing external packages in our code then? Two choices (assuming external library listed under Depends):
  1. Use the `somepackage::somefunction()` notation
  2. Tell R that we're going to import this function by adding `#' @importFrom somepackage somefunction` to docstring, and then we can use it as-is, i.e. `somefunction()` (still need to add docstring line, even if we have included package in Imports)
- To make our functions available to a user we need to export them: add `#' @export` to the docstring
- Can use this to keep internal helper functions private from the user

Namespace - Summary

- List external packages you use under Imports in the `DESCRIPTION` file
- Put a `#' @export` comment above each of your functions you want to be publicly available
- Reference external functions using either `package::function()` syntax or `#' @importFrom package function` and use as-is
- It seems like a lot of unnecessary work, but it will make your package much cleaner and follow R conventions

Vignettes and Unit Testing

Vignettes

- Vignettes are documents related to your package, typically a user guide
- They are **not** related to the reference manual, this is automatically built from the function documentation
- Historically written in LaTeX (via Sweave), but can use Rmarkdown
- `devtools::use_vignette("<vignette name>")` to create the *vignette* folder and a template vignette
- Can manually knit to preview, but when building package for CRAN submission it will get automatically compiled and placed in inst/ (one of the R CMD CHECKS is that all vignettes compile)
- Take the time to write a strong vignette, and it can form the basis for a later journal submission to JSS

Unit Testing

- Won't spend much time on this as could easily be the subject of a series of seminars!
- Good testing practice will improve your library's robustness and make it easier to spot bugs
- Unit testing involves passing a set of inputs into a function and verifying that the function behaves as expected:
  - Correct output
  - Extreme inputs (missing values, empty lists, NaNs)
  - Handles errors gracefully
  - Guards against incorrectly specified input
- Unit testing encourages single-use functions
- Ideally, write the test before the code
- Check out Hadley's testthat package
  http://r-pkgs.had.co.nz/tests.html

R CMD CHECK

Passing R CMD CHECK

- Once your package is ready for submission, you must check it passes CRAN's strict criteria: `R CMD CHECK`
- Can run from within RStudio[2], but I'd recommend using `devtools::check()` as this will also build the package for you and cleanup after
- Checks for several things:
  - All required information in `DESCRIPTION` is filled out
  - Package imports are ok
  - All unit tests pass
  - All function documentation is available
  - Package can be built (including vignette compiling) and is < 5MB
  - Package can be installed on a user's machine

[2] Build -> Check

Common reasons for failing

- Namespace issues!
- Not realising some functions aren't `base` and need importing, i.e. `stats::lm()`
- *No visible binding* - when use a column name without the $. Often found when using dplyr/ggplot (use strings instead)
- A function that you don't directly call, but is used by an external function isn't imported. Find them through `R CMD CHECK` and add to `#' @importFrom ...`
- Haven't updated version number or date in `DESCRIPTION`

Submission

- Once package passes `R CMD CHECK` it's time for submission!
- Two options:
  1. Build package (using `devtools::build()` or RStudio button) and manually submit at https://cran.r-project.org/submit.html
  2. Use `devtools::submit()`
- A CRAN moderator will get back to you relatively promptly with either instructions on what needs to be fixed, or to inform you your package has been accepted
- Be patient, I had package rejected twice!

Other resources

- Hadley's Wickham's book *R Packages* is very useful!
- Also available online at http://r-pkgs.had.co.nz/
- Karl Broman's posts
  http://kbroman.org/pkg_primer/pages/cran.html
- François Briatte on submitting
  http://f.briatte.org/r/submitting-packages-to-cran

Any questions?