Chapter 43: Creating packages with devtools

This topic will cover the creation of R packages from scratch with the devtools package.

Section 43.1: Creating and distributing packages

This is a *compact guide* about how to quickly create an R package from your code. Exhaustive documentations will be linked when available and should be read if you want a deeper knowledge of the situation. See *Remarks* for more resources.

The directory where your code stands will be referred as `./`, and all the commands are meant to be executed from a R prompt in this folder.

Creation of the documentation

The documentation for your code has to be in a format which is very similar to LaTeX.

However, we will use a tool named `roxygen` in order to simplify the process:

```
install.packages("devtools")
library("devtools")
install.packages("roxygen2")
library("roxygen2")
```

The full man page for roxygen is available here. It is very similar to *doxygen*.

Here is a practical sample about how to document a function with *roxygen*:

```
#' Increment a variable.
#'
#' Note that the behavior of this function
#' is undefined if `x` is not of class `numeric`.
#'
#' @export
#' @author  another guy
#' @name    Increment Function
#' @title   increment
#'
#' @param x   Variable to increment
#' @return    `x` incremented of 1
#'
#' @seealso   `other_function`
#'
#' @examples
#' increment(3)
#' > 4
increment <- function(x) {
  return (x+1)
}
```

And here will be the result.

It is also recommanded to create a vignette (see the topic *Creating vignettes*), which is a full guide about your package.

Construction of the package skeleton

Assuming that your code is written for instance in files `./script1.R` and `./script2.R`, launch the following command in order to create the file tree of your package:

```
package.skeleton(name="MyPackage", code_files=c("script1.R","script2.R"))
```

Then delete all the files in `./MyPackage/man/`. You have now to compile the documentation:

```
roxygenize("MyPackage")
```

You should also generate a reference manual from your documentation using `R CMD Rd2pdf MyPackage` from a *command prompt* started in `./`.

Edition of the package properties
1. Package description

Modify `./MyPackage/DESCRIPTION` according to your needs. The fields `Package`, `Version`, `License`, `Description`, `Title`, `Author` and `Maintainer` are mandatory, the other are optional.

If your package depends on others packages, specify them in a field named `Depends` (R version < 3.2.0) or `Imports` (R version > 3.2.0).

2. Optional folders

Once you launched the skeleton build, `./MyPackage/` only had `R/` and `man/` subfolders. However, it can have some others:

- `data/`: here you can place the data that your library needs and that isn't code. It must be saved as dataset with the `.RData` extension, and you can load it at runtime with `data()` and `load()`
- `tests/`: all the code files in this folder will be ran at install time. If there is any error, the installation will fail.
- `src/`: for C/C++/Fortran source files you need (using `Rcpp`...).
- `exec/`: for other executables.
- `misc/`: for barely everything else.

Finalization and build

You can delete `./MyPackage/Read-and-delete-me`.

As it is now, your package is ready to be installed.

You can install it with `devtools::install("MyPackage")`.

To build your package as a source tarball, you need to execute the following command, from a *command prompt* in `./`: `R CMD build MyPackage`

Distribution of your package
Through Github

Simply create a new repository called *MyPackage* and upload everything in `MyPackage/` to the master branch. Here is an example.

Then anyone can install your package from github with devtools:

```
install_package("MyPackage", "your_github_usename")
```

Through CRAN

Your package needs to comply to the CRAN Repository Policy. Including but not limited to: your package must be cross-platforms (except some very special cases), it should pass the `R CMD check` test.

Here is the submission form. You must upload the source tarball.

Section 43.2: Creating vignettes

> 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.

Vignettes will be created entirely in markdown.

Requirements

- Rmarkdown: `install.packages("rmarkdown")`
- Pandoc

Vignette creation

```
devtools::use_vignette("MyVignette", "MyPackage")
```

You can now edit your vignette at `./vignettes/MyVignette.Rmd`.

The text in your vignette is formatted as Markdown.

The only addition to the original Markdown, is a tag that takes R code, runs it, captures the output, and translates it into formatted Markdown:

````
```{r}
# Add two numbers together
add <- function(a, b) a + b
add(10, 20)
```
````

Will display as:

```
# Add two numbers together
add <- function(a, b) a + b
add(10, 20)
## [1] 30
```

Thus, all the packages you will use in your vignettes must be listed as dependencies in `./DESCRIPTION`.