Introduction to R Package Development

Dirk Eddelbuettel
dirk.eddelbuettel@R-Project.org
edd@debian.org
@eddelbuettel

Big Data and Open Science with R
Warren Center for Network and Data Sciences
University of Pennsylvania, Philadelphia, PA
21 November 2014
Outline

1 Why
> fortunes::fortune(92)

##
## If you don't go with R now, you will someday.
## -- David Kane (on whether to use R or S-PLUS)
## R-SIG-Finance (November 2004)
A language and an environment (cf R FAQ)

Has forever altered the way people analyze, visualize and manipulate data (cf 1999 ACM citation)

A vibrant community and ecosystem: CRAN + BioConductor provide > 6k packages that “just work”

The lingua franca of (applied) statistical research

Reliable cross-platform + cross-operating system

Yet occasional challenges of getting R and code to collaborators, students, ...
Key points from the previous slide:

- **community**: CRAN / packages part of R’s success
- **cross-platform / cross-OS**: packages are portable
- **getting R [...] code to collaborators**: distribution
More key points:

- **reproducibility**: aided greatly by identifiable package versions
- **version control**: learn about git (or svn)
- **quality control**: package creation / update is QA
Moreover:

- **have impact**: write software others use
- **software is the new publication**: name five recent papers, or name five recent packages...
- **save yourself time**: your own use of your own code is eased
Minimal

- R Tools
- pkgA: Very Basic
- pkgB: R Code
- pkgC: Rd Documentation
The Key R Interface to Packages

R CMD build someDirectory to create a package
R CMD check somePackage_1.2-3.tar.gz to check a package
R CMD INSTALL somePackage_1.2-3.tar.gz to install a (source) package
Create a directory and file `pkgA/DESCRIPTION`:

Package: pkgA  
Type: Package  
Title: A First Test  
Version: 0.0.1  
Date: 2014-11-15  
Author: Dirk Eddelbuettel  
Maintainer: Dirk Eddelbuettel <edd@debian.org>  
Description: A minimal package  
License: GPL (>= 2)

Run `R CMD build pkgA` and `R CMD check pkgA_0.0.1.tar.gz`. 
Do It By Hand – Once

Note the OKs – and absence of NOTE, WARNING or ERROR.
The package build told us it added a file `NAMESPACE`:

```r
# Default NAMESPACE created by R
# Remove the previous line if you edit this file

# Export all names
exportPattern(".")
```

These lines are now mandatory and control want you “export” and “import”.
About DESCRIPTION

There is more than we have time to discuss now:

Author: Give credit where credit is due
Maintainer: Generally you, with a valid email address
Version: Semantic Versioning in the form a.b.c is popular, and sensible.
License: Matters, and worth giving it some thought.
Depends: Important when you have dependency
Imports: Dependency as import() or importFrom() – no time for this today
LinkingTo: No time to dive into this today
OS_type: Occassional restriction
SystemRequirements: For special needs
So the initial attempt worked and created a valid – but useless – package.

Now add a directory \texttt{R/} and a file with a function or two.

Suggestion: Compute tail quantiles of a vector.
Something like this in a file `R/myqs.R`:

```r
## simple function to return quantiles of vector
myqs <- function(x, 
                 at=c(0.01, 0.05, 0.10, 0.50, 
                      0.90, 0.95, 0.99)) {

  ## should do some sanity checks on x here

  res <- quantile(x, probs=at)
}
```
Do It By Hand – Once

Note the WARNING.
Now we need to add a help page in Rd format.

By convention a file in `man` with the same name as the corresponding R code, ie `man/myqs.Rd`. 
Do It By Hand – Once

Better tools are available as eg roxygen2 – covered later.
Do It By Hand – Once

Once again full of OKs.
Outline

3 Using Tools
- Overview
- pkgD: package.skeleton()
- pkgE: kitten()
Helpers For Creating a Package

package.skeleton() main worker, has warts; also called by RStudio

kitten() corrects issues with package.skeleton()

create() an alternative from devtools (which I do not use much)
Using package.skeleton()

> `setwd("code")`
> `package.skeleton("pkgD")`

## Creating directories ...  
## Creating DESCRIPTION ...  
## Creating NAMESPACE ...  
## Creating Read-and-delete-me ...  
## Saving functions and data ...  
## Making help files ...  
## Done.  
## Further steps are described in './pkgD/Read-and-delete-me'.

Looks good, but `R CMD check ...` fails and dies.
Using `kitten()`

```r
> setwd("code"); library(pkgKitten); kitten("pkgE")

## Creating directories ...  
## Creating DESCRIPTION ...  
## Creating NAMESPACE ...  
## Creating Read-and-delete-me ...  
## Saving functions and data ...  
## Making help files ...  
## Done.  
## Further steps are described in './pkgE/Read-and-delete-me'.  
## Adding pkgKitten overides.  
## Deleted 'Read-and-delete-me'.  
## Done.  
##  
## Consider reading the documentation for all the packaging details.  
## A good start is the 'Writing R Extensions' manual.  
## And run 'R CMD check'. Run it frequently. And think of those kittens.
```
Checking kitten()

Once again full of OKs.
The header of our `myqs()` function could look like this:

```r
# A simple demo function which returns quantiles

# This is just an example.
# @title Simple quantile calculator
# @param x A vector for which quantiles are to be calculated
# @param at A vector with p-values for the desired quantiles
# @return A named vector with the desired quantiles.
# @see \link[stats:quantile]{quantile}
# @references None
# @author Dirk Eddelbuettel
# @examples
# set.seed(123)  # be reproducible
# x <- rnorm(1000)
# myqs(x)

myqs <- function(x, at=c(0.01, 0.05, 0.1, 0.5, 0.9, 0.95, 0.99)){
    # should do some sanity checks on x here
    res <- quantile(x, probs=at)
}
```
Using roxygen2

The usage is simple: call `roxygenize()`.

```r
> setwd("code/pkgCroxy")
> library(roxygen2)
> roxygenize(".", roclets="rd")
```

which writes `myqs.Rd` for us.
Outline

4 Special Topics

- Data
- Anything
- Unit Tests
- Vignettes
- Compiled
We can include data sets in packages.

Each data set should have a manual page as well, and there is roxygen2 support.

A (very) recent example is the sp500 dataset in the lltf package by Hadley.

Documentation is in Chapter 1.1.6 of Writing R Extensions.
See `pkgEdata` – a copy of `pkgE` with a `data/` directory.

It contains one data set, one helper function and documentation for the data set.

```r
#' @name somedat
#' @title somedat - fake data set as an example
#' @description This data set contains a columns of data, a time-trend variable foo and a noise variable bar
#' @docType data
#' @usage data(somedat)
#' @source Made-up by internal function \code{.dataCreation()}
#' @author Dirk Eddelbuettel
#' @keywords datasets

NULL

.dataCreation <- function() {
  # a boring fictious data.frame
  set.seed(124)
  N <- 100
  somedat <- data.frame(date=as.Date("2001-01-01") + 0:100,
                         foo=100 + seq(1,N)*0.25 + rnorm(N),
                         bar=runif(100)*0.5 + 50)
  save(somedat, file="../data/somedat.RData")
  invisible(NULL)
}
```
## R Code

```r
> data(somedat, package="pkgEdata")
> head(somedat)
```

```
##     date     foo     bar
## 1 2001-01-01 98.86493 50.15971
## 2 2001-01-02 100.53832 50.43209
## 3 2001-01-03 99.98697 50.06274
## 4 2001-01-04 101.21231 50.46950
## 5 2001-01-05 102.67554 50.43260
## 6 2001-01-06 102.24448 50.15180
```
We can include other files in packages as well.

By convention, each file or directory below `inst/` is shipped “as is”. This allows us to access installed files via `system.file`.

A common special case is including header files in `inst/include`, example applications in `inst/examples` or unit tests in `inst/unitTest` or `inst/tests`.

Other useful cases are helper scripts in other languages (Perl, Python, ...).
We include a shell script `inst/scripts/silly.sh`. It is not important what it does – but that we can call it.

```bash
#!/bin/bash

## this is just an example in which we simply output
## data to stdout -- which is "fixed".
##
## a real script would potentially do some work, maybe
## work with command-line arguments etc pp -- but our
## focus here is on the R side of things
##
## also worth reiterating that this could be a Perl,
## Python, Ruby, Node/JS, ... "whatever" script. The
## only thing that matter is that we should be able to
## invoke it on each platform, which may be easiest for
## shell. So this is shell.

cat <<<EOF
date,foo,bar
2001-01-01,10,12
2001-02-01,9,13
2001-03-01,11,12
2001-04-01,12,14
2001-05-01,13,15
2001-06-01,14,17
EOF
```
## Example of calling a script via system

```r
otherViaSystem <- function() {
  path <- system.file("scripts", "silly.sh",
                     package="pkgEother")

  cmd <- paste("sh", path)
  res <- system(cmd, intern=TRUE)
}
```

@title Example of using system on package-supplied script
@return Several lines of text
@author Dirk Eddelbuettel
We can run this:

```r
> library(pkgEother)
> head(otherViaSystem())
```

```r
## [1] "date,foo,bar"  "2001-01-01,10,12"  "2001-02-01,9,13"
## [4] "2001-03-01,11,12" "2001-04-01,12,14" "2001-05-01,13,15"
```
# Example of calling a script via `pip`

Uses `system.file()` to portably obtain the path of a shell script and uses `pipe()` to execute it, using the command output as input to read.

@title Example of using pipe on package-supplied script
@return A data.frame read from the output
@author Dirk Eddelbuettel

```r
otherViaPipe <- function() {
  path <- system.file("scripts", "silly.sh",
                      package="pkgEother")
  cmd <- paste("sh", path)
  res <- read.csv(pipe(cmd))
}
```

---

R Packaging
We can run this as well:

```r
> library(pkgEother)
> head(otherViaPipe())
```

```
##    date  foo  bar
## 1 2001-01-01  10  12
## 2 2001-02-01   9  13
## 3 2001-03-01  11  12
## 4 2001-04-01  12  14
## 5 2001-05-01  13  15
## 6 2001-06-01  14  17
```
Adding unit tests may be one of the best way to ensure quality.

In a nutshell, it means adding short functions which test *invariants*.

Given input, and a function under consideration, compared the generated output to the expected value(s).

This is supported by several packages, notable *RUnit* and *testthat* but we probably do not have time to dive into this.
Shipping Tests: Simplest approach

Place a file `foo.R` in the directory `tests/`, run `R CMD check` on the package and copy the resulting `foo.Rout` as `foo.Rout.save` in `tests`.

```r
> library(pkgEsimpletests)
>
> set.seed(123)  # be reproducible
> x <- rnorm(1000)  # some data
> res <- myqs(x)  # run our function of interest
> print(res)  # print result
```
Below is the corresponding `foo.Rout.save` – R will during check time compare its output to the freshly generated one.

```r
R version 3.1.2 (2014-10-31) -- "Pumpkin Helmet"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: i686-pc-linux-gnu (32-bit)
[...]
>
> library(pkgEsimpletests)
>
> set.seed(123) # be reproducible
> x <- rnorm(1000) # some data
> res <- myqs(x) # run our function of interest
> print(res) # print result
1%   5%   10%   50%   90%   95% 99%
-2.158176203 -1.622584310 -1.267328289 0.009209639 1.254751947 1.676133871 2.397645689
>
> proc.time()
user  system elapsed
0.312 0.352 0.271
```
RUnit is one of several packages supporting unit tests. When used, we need to add Suggests: RUnit to the NAMESPACE file.

We place a (essentially fixed) script calling the RUnit testrunner engine in tests/

```r
stopifnot(require(RUnit, quietly=TRUE))
stopifnot(require(pkgEunittests, quietly=TRUE))
set.seed(42)  # Set a seed to make the test deterministic

## Define tests
testSuite <- defineTestSuite(name="pkgEunittests Unit Tests",
dirs=system.file("tests", package="pkgEunittests"),
testFuncRegexp  = "^[Tt]est+")

tests <- runTestSuite(testSuite)  # Run tests
printTextProtocol(tests)  # Print results

# Return success or failure to R CMD CHECK
if (getErrors(tests)$nFail > 0) stop("TEST FAILED!")
if (getErrors(tests)$nErr > 0) stop("TEST HAD ERRORS!")
if (getErrors(tests)$nTestFunc < 1) stop("NO TEST FUNCTIONS RUN!")
```
Shipping Tests: Using RUnit

We also place files matching `runit.*.r` in `inst/tests`.

```r
#.setUp <- function() { }
# can run some code needed below here, eg a database connection,
#.tearDown <- function() { }
# similar function to clean up at end

test01leftTail <- function() {
  set.seed(123)        # be reproducible
  x <- rnorm(1000)     # some data
  res <- myqs(x)       # run our function of interest
  comp <- quantile(x, probs=c(0.01, 0.05, 0.10, 0.50, 0.90, 0.95, 0.99))
  checkEquals(res[1], comp[1], msg="checking 1%-tile")
  checkEquals(res[2:3], comp[2:3], msg="checking 5% and 10%-tile")
}

test02rightTail <- function() {
  set.seed(123)        # be reproducible
  x <- rnorm(1000)     # some data
  res <- myqs(x)       # run our function of interest
  comp <- quantile(x, probs=c(0.01, 0.05, 0.10, 0.50, 0.90, 0.95, 0.99))
  checkEquals(res[7], comp[7], msg="checking 99%-tile")
  checkEquals(res[5:6], comp[5:6], msg="checking 90% and 95%-tile")
}
```
Vignettes – documentation in pdf or html format – can be added the via the directory `vignettes/`. They may also require an entry `VignetteBuilder:` in `DESCRIPTION`.

Formats are either generally markdown or latex – and either form can incorporate embedded R code (and even code in other languages).

Yihui will cover the markdown variant tomorrow morning.
Shipping Vignettes: Example

```
edd@don:~$ git samplecode/penn-2014-11
edd@don:~$ cat pkgEvignette/vignettes/pkgEvignette.Rnw
\documentclass[litp]{article}
%VignetteIndexEntry{pkgEvignette}
%VignetteKeywords{package,example}
%VignettePackage{pkgEvignette}
\usepackage{margin=lin,paper=letterpaper}{geometry}
\usepackage[T1]{fontenc}
\usepackage[pslatex] \ just like AJournal
\usepackage{palatino,mathpazo}
<<echo=FALSE,print=FALSE>>=
prettyVersion <- packageDescription("pkgEvignette")[["Version"]]
prettyDate <- format(Sys.Date(), "%B %e, %Y")
@

\author{Some Author}
\title{pkgEvignette: A small Example}
\date{\pkg{pkgEvignette} version \texttt{\#X expres{prettyVersion} as of \texttt{\#Xexp{prettyDate}}}
\begin{document}
\maketitle
\abstract{
\indent Not much to say at this point.
}
\section{Some Text}
Some code:
<<sample,fig=TRUE>>=
set.seed(123)
x <- cumsum(rnorm(100))
plot(x, type='l', main="A random walk")
@
Code in a line: \texttt{\#Xexp{sqrt(3)}}.
\end{document}
edd@don:~$ git samplecode/penn-2014-11
```
The **RcppArmadillo** package (discussed more tomorrow) has a variant **RcppArmadillo.package.skeleton()**:  

```r
> setwd("code")
> library(RcppArmadillo)
> RcppArmadillo.package.skeleton("pkgF")
```
Using RcppArmadillo

Also full of OKs. More on this tomorrow.
Other resources on packaging

*Writing R Extensions* by the R Core team  authoritative but somewhat terse

*R packages* by H Wickham  book in progress, thorough but too opinionated on *devtools* etc

*Building R Packages* by D Diez  nice slide deck, though getting a little dated

*Developing R packages* by J Leek  excellent, but somewhat BioConductor focussed

*Creating R Packages* by F Leisch  classic, thorough, also covers OO in R, a little dated too

*Write your own Package* by Stat545/UBC  very new, *devtools*-centric as well.
Other resources on packaging

And of course maybe *the* best resource:

- **CRAN** with over 6000 packages
- **BioConductor** is also a very good source, with a strong development culture and many fine tutorials
- **GitHub** now contains mirrors of CRAN and more, and can be searched.
Topics not covered

- Version control systems [VCS] (eg `git`, `svn`, ...) – highly recommended
- Continued Intgegration [CI] (ie Travis, Jenkins, ...) working with VCS
- Reproducible research and attempts snapshot installations such as `packrat`
- Docker (and our Rocker project) containers for R deployment, testing, reproducibility, ...
- Plus much, much more.
So go on ...

While there is always more to learn, and more details to uncover – you should now have a basis to start from.

So package on!