Building R Packages

An Introduction

David Diez
Biostatistics Dept
Harvard SPH

Original version and source

Original author: David M Diez

The production of these slides was funded by

- NIH/NCI P01CA134294 (Lin): Statistical Informatics for Cancer Research. This project was supported by Award Number P01 CA134294 from the National Cancer Institute. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Cancer Institute.

Why build an R package?

Accessible

- Functions and objects contained in a package and installed on a machine can be easily loaded:
  `> library(myPackage)`
- Many R users develop their own functions that they use regularly
- Even for a sole user, putting code into a package can be worthwhile

Reliable

- Documentation structure is familiar, and it is pretty easy to edit
- Testing can be built into the package itself

Clarity

- The process of organizing code and data into a package requires a project to become organized
- The result is less ambiguity about project goals and greater clarity about how the project will be completed

Why every grad student should build a package

Three important blocks of modern statistics

- Math: methodological development
- Science: applications to real world problems
- Computing: make statistical methods accessible

Fulfilling the computing block

- Traditional research focuses more on methods and applications
- Building an R package suggests competence in computing

Employability

- Not many grad students build an R package
- Display an ability to generalize code and make it user-friendly
- Potential employers can better understand what you've worked on

Important software principles

Goal or mission

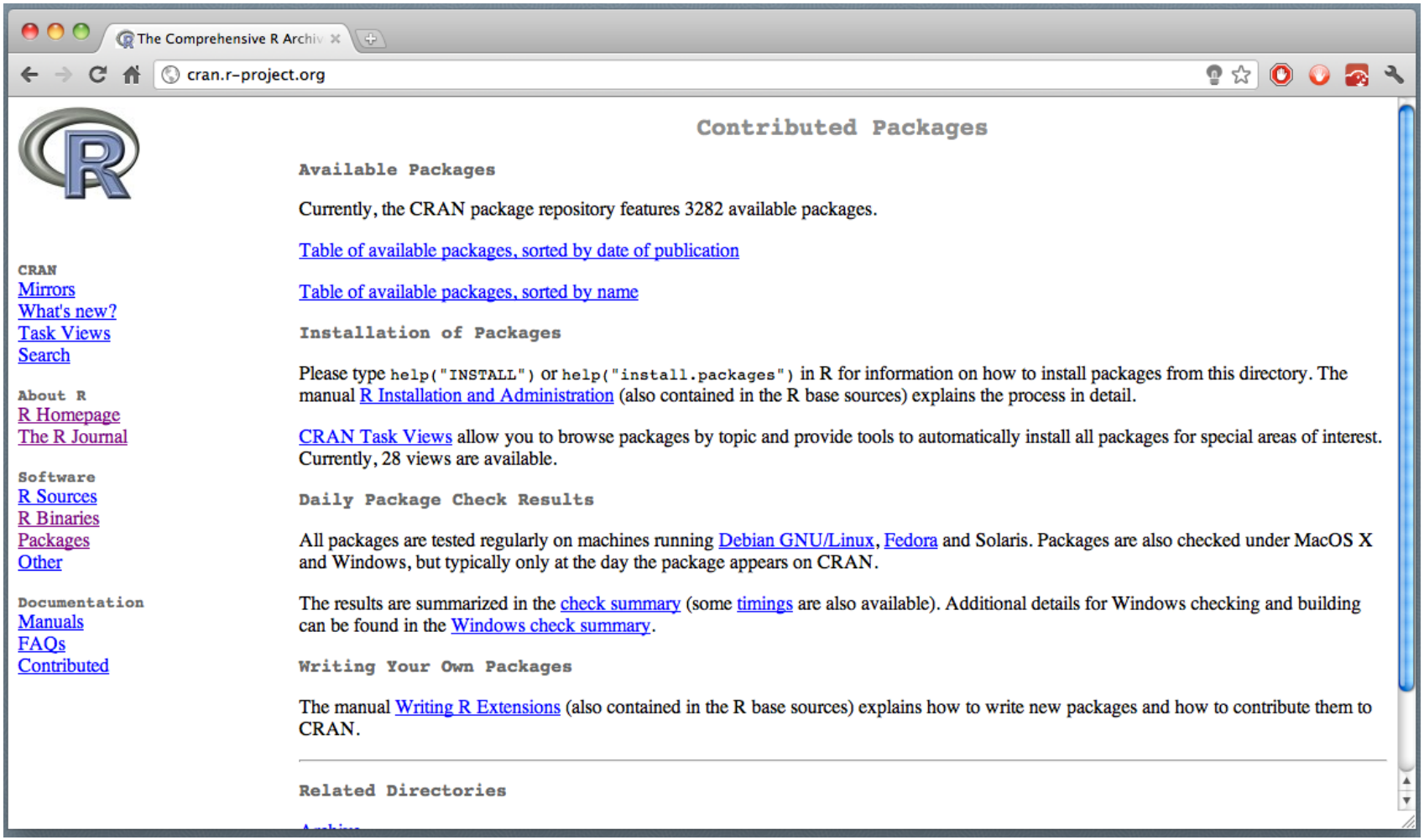
- The process of organizing code and data into a package requires a project to become organized
- The result is less ambiguity about project goals and greater clarity about how the project will be completed

Coding principles

- Make high-quality software
- Implement clean coding practices so the code can be adequately reviewed and verified
- Provide helpful documentation

Sharing data, functions, and an analysis online

Currently, CRAN features **3282** *available packages* (as of 9/15/2011, up from 2564 on 10/5/2010).



What are all these packages?

Statistical and other methods

- Many R packages make accessible previously or newly developed statistical methods
- Graphical functions, complex numerical techniques, making it easier to work with big data sets, etc.

Open research

- Publishing a paper for a new method does not make the technique open and accessible
- Ideally, researchers could try out a new method without prohibitive effort, i.e. not have to code it themselves

Data

- Sharing old, new, simulated, or research data sets
- Many of the best packages have both methods and data

3282 packages and counting

Initially daunting

- If there are already so many packages, is there room for one more?
- Some might say the same about research: There are so many statistical methods, so can I really develop something both novel and helpful?
- The answer to each question better be yes for the sake of anyone wanting a PhD

Keep an eye out

If you are performing raw coding in R, one of the following is true:

- You are ignoring prebuilt functions in R or in an available package
- The method is too user-specific to have a general function
- This may be a place for a new R package

Ultimate goal

- Build a package to fulfill a need

Considerations

- The span of R users is wide: applied, software development, visualization, teaching, etc.
- Even if a method is already available, it doesn't mean it was written well or is accurate
- Some R user groups are ignored: find a niche

Create a mission

Ask important questions from the start

- What are you good at?
- What is needed?
- What part of that gap can you fill?

Identify a target audience

- Beginner or advanced?
- Researcher, student, or teacher?
- Do your target users have lots of data or none at all?

Avoid wasted time

- When a package isn't needed, identify this early on
- Sketch out what would be included in the ideal package that accomplishes your mission
- Identify at what stage the package (possibly preliminary) could and/or should be released

Example: stockPortfolio

Offer a "starter" package for financial analysts who want to get into statistical modeling with R but have little background in statistical finance and/or R

What is needed: a logical procedure to familiarize the process of collecting data, modeling, and obtaining results from models:

(1) Get the data

```
> tickers    <- c('C','BAC', 'WFC', 'GS')
> financials <- getReturns(tickers, start='2004-01-01',
+                          end='2008-12-31')
```

(2) Build the model

```
> sm         <- stockModel(financials, model='CCM')
```

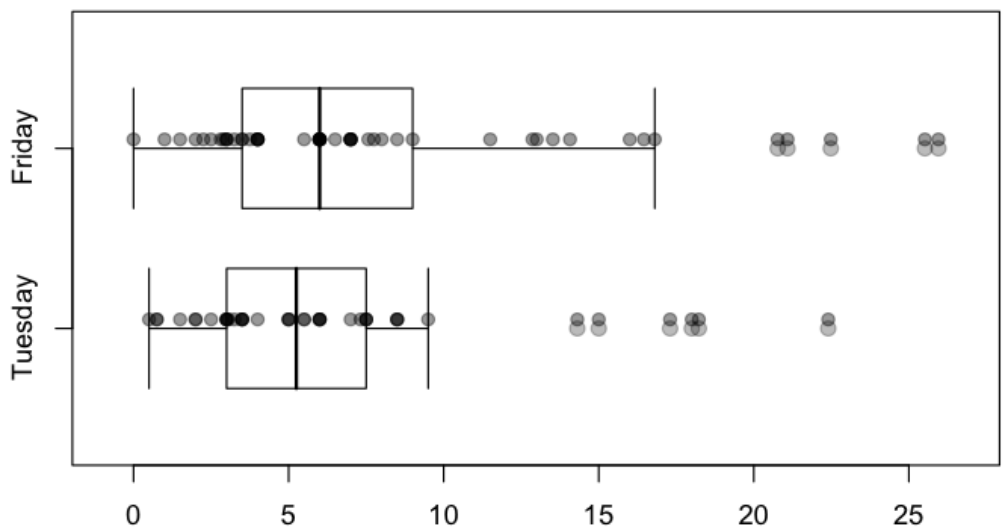
(3) Obtain the optimal portfolio

```
> opSM       <- optimalPort(sm)
```

Example: openintro

Provide data and simple graphical functions for reproducing results and figures in the **OpenIntro Statistics** textbook

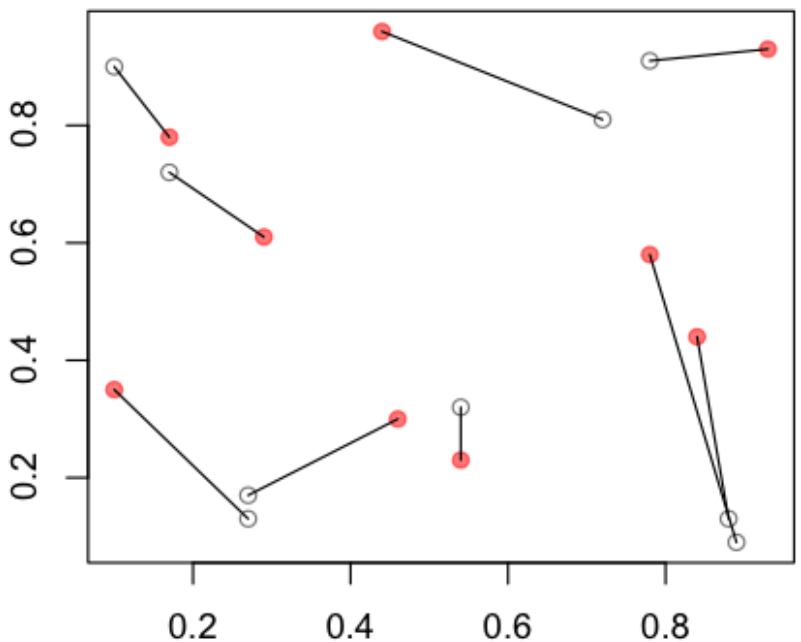
```
> data(tips)
> par(mfrow=c(1,1))
> boxPlot(tips$tip, tips$day, horiz=TRUE,
+           key=c('Tuesday', 'Friday'))
> dotPlot(tips$tip, tips$day, add=TRUE,
+           at=1:2+0.05, key=c('Tuesday', 'Friday'))
```



Example: ppMeasures

Provide basic functions for implementing new methods and reproducing major results from dissertation work

```
> data(pattEx2)
> x <- pattEx2[pattEx2[,1] == 1,c(2,3)]
> y <- pattEx2[pattEx2[,1] == 2,c(2,3)]
> (hold2 <- stDist(x, y, 2))
[1] 5.54
> summary(hold2)
Algorithm: IMA
Max branch: 4
9 points were matched
Distance: 5.54
> plot(hold2)
```



Find what already exists

Examine CRAN

- `cran.r-project.org`
- Look for similar topics
- Identify the audience of other packages
- Check if overlapping packages are adequate

Other repositories to check/consider

- R Forge: `rforge.net`
- Bioconductor: `bioconductor.org`
- This list is not exhaustive!

Writing and generalizing code

Balance simplicity with complexity

- Want to offer diverse set of options
- It should be in a form that it will not be too complex
- Target audience is a guide to the right balance
- Offer many arguments and choose appropriate defaults

Example: stockPortfolio

- The models implemented are generally basic
- Intended for folks breaking into stock modeling, i.e may not be familiar with R
- Result: 3-step procedure for implementing any of the models from only a few functions
- Advanced options are made available for users who are interested in learning more

General R coding advice

Performance

- Initialize an entire object rather than grow it slowly
- Compute unchanging values only once

Functionality

- Choose variable and function names carefully
- Create default values and use `...` in functions when it's helpful
- Outputting a list? Give each list item a name

Aesthetics

- Align assignment characters
- Use tabs and white space for alignment or when it is meaningful
- If including comments, do so in a style that is not obstructive
- Avoid all caps
- No more than one assignment per line of code

Evaluating and re-evaluating

Build a foundation of diverse examples

- Look for ways to improve speed, accuracy, and usability

Sufficiently general

- Have a colleague/friend look at the function
- Does it work well for the original problem?
- Is it easy to apply to similar problems?
- Can it be further generalized, or would that be too confusing?

Example

- ~~Rome~~ `glm` wasn't built in a day
- Developers could have made one function for each scenario
- Instead they simplified everything: different scenarios are addressed by modifying arguments, and these arguments have good defaults

Picking data sets

Which examples highlight the package?

- If the package is function-centric, choose examples that highlight the performance and graphics
- If your method might be known to be a poor choice in some instances, it would be helpful to point this out to researchers, possibly with an example
- For data-centric packages, use basic functions to show off the data
- Be clear if data are not real or were collected in a haphazard fashion
- Real data are strongly preferred

Common knowledge worth repeating

- Don't release data unless you have permissions to or the data are public

Why use classes?

Classes make it easy to apply general R functions

- We can change the class of an object in R to be `'ourClass'`: `class(myObject) <- 'ourClass'`
- Next we build special methods, e.g.
  - `print.ourClass`
  - `summary.ourClass`
  - `plot.ourClass`
- When we apply `plot` to a function of class `'ourClass'`, R actually applies the function `plot.ourClass`

Classes are useful for communication and experimentation

- Allowing the user to connect new functions with old functions is helpful

Downside: classes can mask what is actually contained in an object

How to create classes

An object's class can be *assigned*:

```
stockModel <- function (stockReturns, # other args omitted
){
  # some code omitted
    tM         <- list()       # tM = The Model to be returned
    class(tM) <- "stockModel" # assign the class !
    tM$model  <- model[1]


  # lots of amazing R code


    return(tM)
}
```

Building methods

```
print.stockModel <- function (x, ...){
    cat("Model:", x$model, "\n")
    cat(x$n, "observations, each one", x$period, "apart\n")

  # some code omitted

    colnames(hold) <- theNames
    temp           <- format(hold, digits = 2, scientific = FALSE)
    print(temp)
}

plot.stockModel <- function (x, xlab = "Risk", # some args omitted
){
    # code for plotting a stockModel object, x
    # ...
    # want to return some object?
    invisible(objectToReturn)
}
```

Considerations

Pros of classes

- Users can apply familiar R functions to new objects
- Allows output to be formatted for user digestion
- Saves the user time in finding or visualizing important information

Cons of classes

- Using methods for classes – especially for `print` – takes the user one step away from the true R object
- Some users are unsure how to explore all the attributes of new objects

General tip: see what's in a list object via subsetting or `str`:

```
> objName[1:5] # prints first five list items
> str(objName) # prints summary information
```

Overview

Step 1: Create the package files

- Package all data and objects from an R session:
  `> package.skeleton('packageName')`
- See `?package.skeleton` for additional options

Step 2: Edit the package files

- The `DESCRIPTION` and help files (`man` > `.Rd`) need to be filled in
- Changes to functions should be done directly to the package files
- C or other non-R source code is placed in its own `src` folder

Step 3: Build, check, and install the package

- Run a few Unix commands to build, check, and install the package
- Usually errors arise when checking the package, so return to step 2 as needed

Step 1: The package files

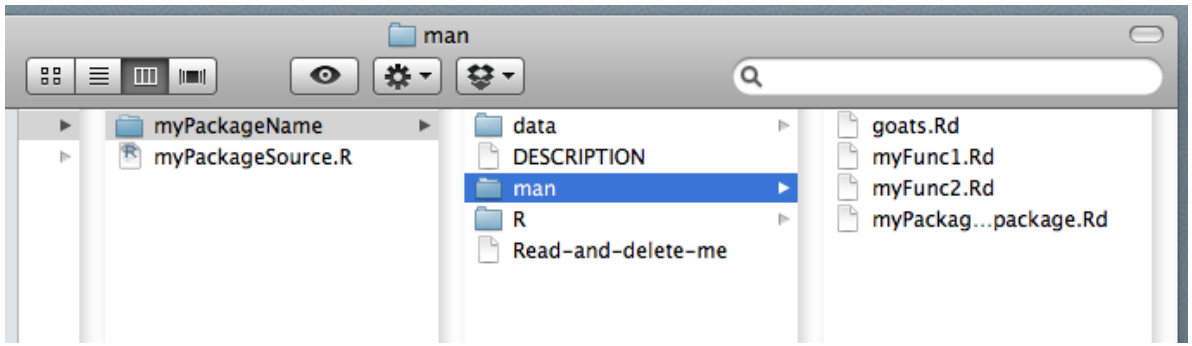
Create the package files:

```
> myFunc1 <- function(x){ }
> myFunc2 <- function(xy, o5){ }
>
> goats <- data.frame(beards = rexp(50),
+                     tails  = rnorm(50, 10))
>
> package.skeleton("myPackageName")
Creating directories ...
Creating DESCRIPTION ...
Creating Read-and-delete-me ...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in './myPackageName/Read-and-delete-me'.
>
```

Step 1: The package files

Folders within the newly created `myPackageName` folder

- `data` – Contains `.rda` files of each data object
- `R` – Contains `.R` files for each function
- `man` – Help files for each function, data set, and the function

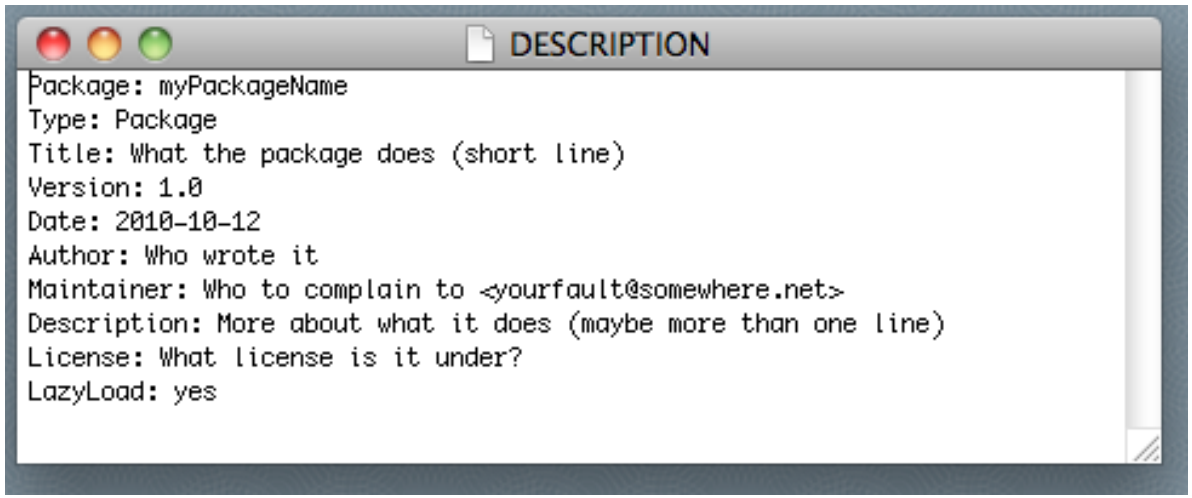


- `src` – Create this folder for any C or FORTRAN source code
- `tests` – Create this folder for any test code
- Other folders with special meanings: `demo`, `exec`, `inst`, `po`

Step 2: Edit the package files

Edit the `DESCRIPTION` file

- Update all information
- Choose your license (e.g. GPL-3)



Step 2: Edit the package files

Edit each help file in the `man` folder

- Use `\code{ }` to write in Courier
- Link to other help files via `\link{ }`: `\code{\link{myFunc2}}`
- May create new help files via `prompt` function in R

myFunc2.Rd

```
\name{myFunc2}
\alias{myFunc2}
%- Also NEED an '\alias' for EACH other topic documented here.
\title{
%%  ~~function to do ... ~~
}
\description{
%%  ~~ A concise (1-5 lines) description of what the function does. ~~
}
\usage{
myFunc2(xy, o5)
}
%- maybe also 'usage' for other objects documented here.
\arguments{
  \item{xy}{
%%     ~~Describe \code{xy} here~~
}
  \item{o5}{
%%     ~~Describe \code{o5} here~~
}
}
\details{
%%  ~~ If necessary, more details than the description above ~~
}
\value{
%%  ~Describe the value returned
%%  If it is a LIST, use
%%  \item{comp1 }{Description of 'comp1'}
%%  \item{comp2 }{Description of 'comp2'}
%% ...
```

Step 2: If documentation is not important

`DESCRIPTION` file

- Choose your license (e.g. GPL-2)

In the `man` folder

- Make sure all help files have some title that is not commented out
- In the package help file (`man > myPackageName-package.Rd`), leave the examples section empty or put in only working R code

Caution: If you don't build adequate help files...

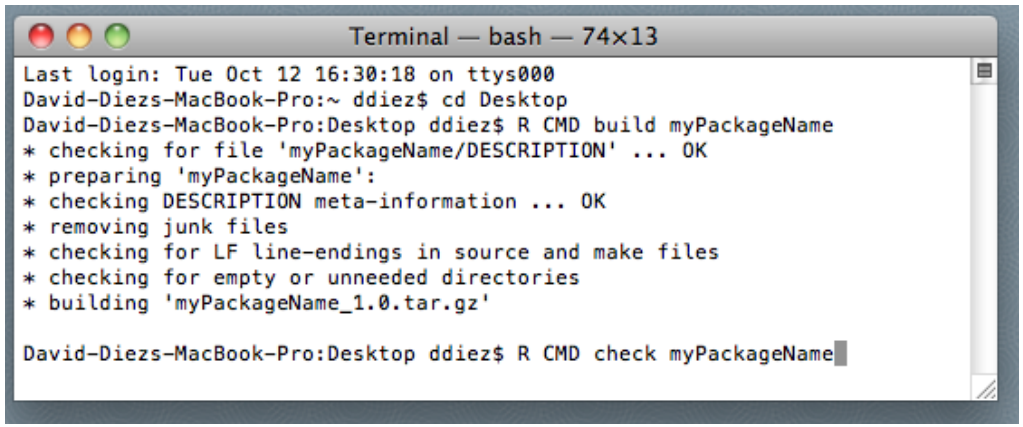
- Will the package be clear when you return to it in a year?
- Is saving time now worth the chance of spending more later?

Step 3: Build, check, and install the package

There are a few ways to do this. One way:

- Drag/drop package file to the Desktop
- Open Unix (e.g. Terminal in Mac OS X), navigate to the desktop, and type

```
R CMD build myPackageName
R CMD check myPackageName
R CMD install myPackageName
```



Step 3: Build, check, and install the package

Remarks (for `check`)

- Warnings and errors are very common in the `check` stage
- Sometimes the package will install even if `check` returns an error
- Package only for personal use? Consider skipping the `check` stage
- CRAN will *not* accept a package that has warnings or errors from `check`

Other useful UNIX commands

`R CMD REMOVE packName`

- Remove a package

`R CMD BUILD --binary packName`

- Creates a binary archive of a package

`R CMD Rd2pdf packName`

- Make a PDF manual for a package

Recap on building the package

Step 1: Create the package files

- Packaging all data and objects in an R session is easy:
  `> package.skeleton('packageName')`

Step 2: Edit the package files

- Fill in `DESCRIPTION` and `man` files
- May edit functions, but make corresponding changes in help files

Step 3: Build, check, and install the package

- If a package is being submitted to CRAN, it *must* pass `check`
- Warning: installing a package will overwrite any previous version of the package

Potential trouble

Packages A and B have different functions but these functions share the same name, `fcnName`

- One of your functions relies on `fcnName` from package A
- If user loads your package (which also loads package A), that user might also load package B
- If your package doesn't have a namespace but relies on `fcnName`, the function from package B might be called instead of the function from package A

Namespaces help prevent such errors

Namespaces

Namespaces manage how the user can interact with a package, and it also facilitates high-level communication among packages

- A `NAMESPACE` file is optional, but if added it goes in the main directory of the package
- Contains instructions for what is imported from other packages
- Describes what files should be easily accessed by other packages

Most packages can get by without a namespace, but occasionally trouble can arise

Tip: use a namespace when publishing a package to CRAN whenever your package relies on another package

Tip: build your namespace after you have stopped adding or removing functions from the package to be released

Submitting to CRAN

Verbatim from CRAN:

To "submit" to CRAN, simply upload to `ftp://cran.r-project.org/incoming` and send email to `cran@r-project.org`. Please do not attach submissions to emails, because this will clutter up the mailboxes of half a dozen people.

Note that we generally do not accept submissions of precompiled binaries due to security reasons. All binary distribution listed above are compiled by selected maintainers, who are in charge for all binaries of their platform, respectively.

Submitting to CRAN

Before submitting

- Install the package on your computer and ensure the help files and examples look proper and run as expected
- Verify one last time that `R CMD check` comes with no warnings or errors

Uploading files

- Use an FTP client to upload files

Keep in mind

- CRAN personel post packages for free, so be especially considerate of their time

Remarks

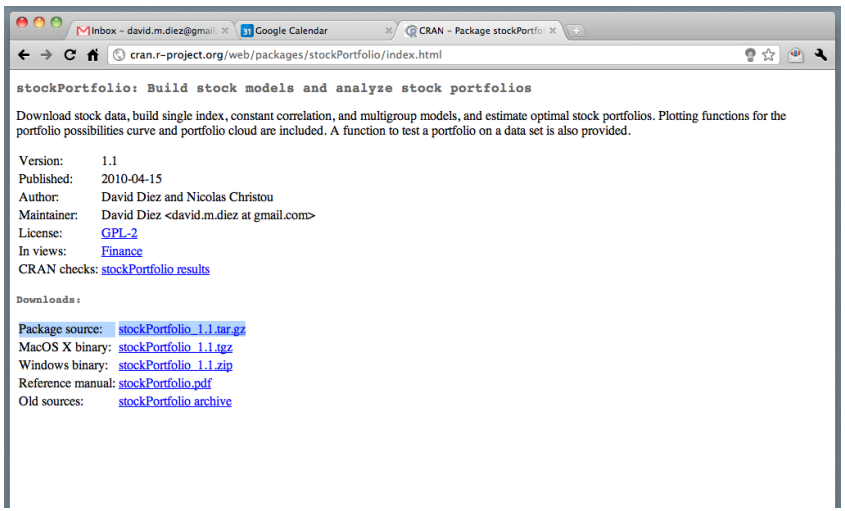
Packages can lead to papers

- Initially a package may provide support for an applied and methodological paper in the name of open research
- A robust package can have its own paper

Two journals to consider, both with free access

- Journal of Statistical Software – `www.jstatsoft.org`
- R Journal – `journal.r-project.org`

Find the source of packages on their CRAN pages



Helpful references

Software for Data Analysis

John Chambers

Springer, 2008

Creating R Packages: A Tutorial

Friedrich Leisch

Department of Statistics
Ludwig-Maximilians-Universität München

R Development Core Team

http://cran.r-project.org/doc/contrib/Leisch-CreatingPackages.pdf

Friedrich.Leisch@R-project.org