

# Building R Packages

## An Introduction

David Diez  
david@openintro.org  
Biostatistics, Harvard SPH

# 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
- Putting code into a package can be worthwhile, even for a sole user

## Reliable

- Documentation structure is familiar, and it is easy to edit
- Basic checks and tests can be automated

## Clarity

- The process of organizing code and data into a package requires a project to become organized and set specific goals

# Sharing data, functions, and an analysis online

CRAN features **3646**, as of 3/2/2012  
(up from 3282 on 9/15/2011 and 2564 on 10/5/2010).

The screenshot shows the CRAN website in a browser window. The address bar displays "cran.r-project.org". The page title is "Contributed Packages". The main content area is titled "Available Packages" and states: "Currently, the CRAN package repository features 3646 available packages." It provides links for "Table of available packages, sorted by date of publication" and "Table of available packages, sorted by name".

Below this, the "Installation of Packages" section explains how to use `help("INSTALL")` or `help("install.packages")` in R, and mentions the manual "Installation and Administration [PDF]". It also notes that "CRAN Task Views" allow browsing packages by topic and that 28 views are currently available.

The "Package Check Results" section states that all packages are tested regularly on machines running Debian GNU/Linux, Fedora, Solaris, Mac OS X, and Windows. It provides a link to the "check summary" and notes that additional details for Windows checking and building can be found in the "Windows check summary".

The "Writing Your Own Packages" section mentions the manual "Writing R Extensions [PDF]" and explains how to write new packages and contribute them to CRAN.

The "Repository Policies" section refers to the manual "CRAN Repository Policy [PDF]" which describes the policies in place for the CRAN package repository.

At the bottom, the "Related Directories" section includes a link to the "Archive" and a note: "Previous versions of the packages listed above, and other packages formerly available."

On the left side of the browser window, there is a navigation menu with the following links: CRAN, Mirrors, What's new?, Task Views, Search, About R, R Homepage, The R Journal, Software, R Sources, R Binaries, Packages, Other, Documentation, Manuals, FAQs, and Contributed.

# What are all these packages?

## Methods

- Facilitate the use of a new or existing statistical technique
- Provide tools for graphics, data exploration, complex numerical techniques, making it easier to work with big data sets, etc.

## Open research

- Researchers publish packages that implement new methods or release data, which supports reproducibility

## Data

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

# Keep an eye out

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

- You are ignoring existing public functions
- The method is too user-specific to have a general function
- This may be a place for a new 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 efficiently, is accurate, or reaches all audiences
- May be preferable to help improve an existing package than to build a new one from the ground-up

# So you want to build a package...

It would be regrettable to spend 100 hours building something that already exists

Review CRAN packages for packages related to your idea

- [cran.r-project.org](https://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](https://rforge.net)
- Bioconductor: [bioconductor.org](https://bioconductor.org)
- This list is not exhaustive!

# So you are going to build a package...

## Mission and goals

- Establish clear aims for the software before starting *and* choose a clear point at which you will publish your work

## Achieve the basics

- Make software that runs, is relatively efficient, and does what it claims
- The software should be intuitive for the target audience

## Good coding practices

- Implement clean coding practices so others can review and verify your work

## Document your work

- Provide helpful documentation with many examples

## Example package: `stockPortfolio`

`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

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

### (3) Obtain the optimal portfolio

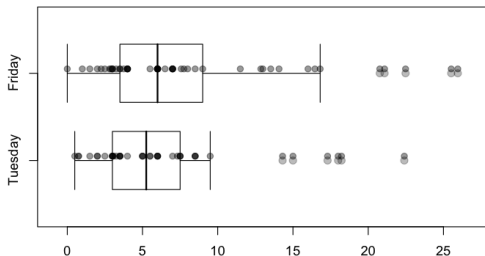
```
> port       <- optimalPort(model)
```



## Example package: `openintro`

`openintro`: Provide data and functions for reproducing results and figures in **OpenIntro Statistics** (open source intro stat textbook)

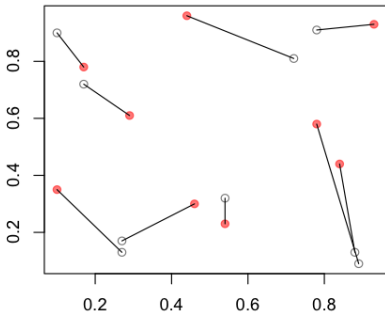
```
> data(tips)
> par(mfrow=c(1,1))
> boxPlot(tips$tip, tips$day, horiz=TRUE,
+         key=c("Tuesday", "Friday"))
> dotPlot(tips$tip, tips$day, add=TRUE,
+         at=1:2+0.05, key=c("Tuesday", "Friday"))
```



## Example package: ppMeasures

**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)]
> (xyd <- stDist(x, y, 2))
[1] 5.54
>
> summary(xyd)
Algorithm: IMA
Max branch: 4
9 points were matched
Distance: 5.54
>
> plot(xyd)
```



# Early planning

## Practical considerations

- Why will this package be important?
- Who will the package serve?
- What supplementals, such as data, are needed?
- What would be included in the ideal package?

## Early code planning:

- To gauge when a first release may be appropriate, what is the least functionality that would still be useful to others?
- What functions will be necessary for this first build of the package?
- What other utilities will be built up in later versions, and how should these affect the structure and functions of the earlier release?

# Planning function details

After roughly laying out function actions and relations

- What arguments will be available in each function?
- What information must be contained in the output of each function?

Complex function output is common

- Are the objects simple enough for users to interact with directly?
- Is there a need for diagnostics, assessment, and exploration of these data objects?
- Could the review of these data objects be streamlined using classes and methods? If so, how should these be structured?

S3 classes and methods are useful in creating a clean user experience for complex data objects, and they are discussed in the next section

# General R coding advice

## Performance

- Initialize an entire object rather than grow it slowly
- Compute unchanging values only once (don't recompute in a loop!)

## Functionality

- Choose variable and function names carefully
- Create helpful, robust default values in functions
- 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

# Evaluating and re-evaluating

## Build a foundation of diverse examples

- Use test cases to assess accuracy
- Using `Rprof`, `Sys.time`, or `system.time`, identify sections of code that offer meaningful opportunities for efficiency improvements

## Sufficiently general

- Does it work well for the original problem?
- Is it easy to apply to similar scenarios and data?
- Are there related settings to which it could be extended?  
(Answering *yes* does not imply the extension must, or even should, be made.)

# Picking data sets

Always try to include data in a package

Which examples highlight the package?

- If the package is function-centric, choose examples highlighting performance and graphics
- If the statistical or computational method performs poorly in some instances, make this clear to researchers, possibly with an example
- For data-centric packages, use standard plotting 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, but simulated data are better than no data

# Classes and methods

Classes and methods encourage and allow users to connect old, familiar functions with new objects

- A **class** is a set of objects that share specific attributes and a common label
- Example: an object of class `"lm"`, generated from the `lm` function, is really just a list with some specific attributes
- With S3 classes in R, we can easily change the class associated with an object or create an entirely new class
- A **method** is a name for a function or action that can be applied to many types of objects
- Examples of methods: `print`, `summary`, `plot`, `predict`
- When we create a new, complex data object from a new function, creating a new class with methods can drastically improve the usability of the function and results



# How to create a new S3 class

To learn the class of an object, apply the `class` function:

```
> x <- list(beard=TRUE, legs=4, tails=1)
> class(x)
[1] "list"
```

We can also use `class` to change an object's class:

```
> class(x) <- "goat"
> class(x)
[1] "goat"
```

# How to create a new S3 class

Usually an object's class is changed before the end-user ever sees it

To create a new class, simply assign a new class to an object before returning it from a function

Example: the `lm` function

- The `lm` function outputs an object of class `"lm"`, which is really just a list with its class changed
- The strategy: initialize the object to be returned, immediately change the initialized object's class, and then continue to add on attributes as needed

**Warning.** It is possible to assign an existing class (e.g. `"lm"`) to a new object, but this generally creates problems if the object doesn't match the structure of other objects in that class

## Building a method (example)

Suppose we have a method, say `print`, that we would like to customize for the new `"goat"` class, then we build a new function called `print.goat`:

```
> print.goat <- function (x, ...){
+   cat("Number of legs:", x$legs, "\n")
+   cat("Number of tails:", x$tails, "\n")
+   y <- ifelse(x$beard, "This goat has a beard", "")
+   cat(y, "\n")
+ }
>
> x   # same as print(x)
Number of legs: 4
Number of tails: 1
This goat has a beard
>
```

# Making more methods

## Generalizing

- Consider a method called `Method` and a new class called `"Class"`
- Suppose we want to allow users to apply `Method` to an object of class `"Class"`
- We create a new function called `Method.Class`, which R will then invoke whenever `Method` is applied to an object of class `"Class"`
- Recall: to construct the `print` method for the `"goat"` class, we made a function called `print.goat`

## Complex objects can and should work with a variety of familiar methods

- Specify a `summary` for a new, complex object of class `"Class"` by writing a new function called `summary.Class`
- Similarly, if appropriate, make a custom method of `plot` for an object of class `"Class"` by creating `plot.Class`

# 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 will be unsure how to explore all the attributes of new objects

General tip: learn about an R object by applying `str`

```
> str(objName) # prints summary information
```

# Prerequisites

Knowing these functions will be useful

`save(..., file="filename.rda")`

- Save specific R objects to a file

`getwd()`

- Learn an R session's current working directory

`prompt(object)`

- Generate a help (`.Rd`) file for an R object, usually for a data object or function that is being added to an existing package

# When to start building

## Existing functions

- Package existing functions immediately to facilitate documentation and access
- May later remove deprecated functions or add new functions (the same is true of data)

## Upcoming projects

- Even if no code or data exist, initialize a package for the project
- Save future functions within the package, and add documentation files once a function's name and arguments take form
- Overhauling a function within a package is not overly complex, so don't hesitate to document a draft of the function

# Overview

## Step 1: Create the package files

- Load the new package's data objects and functions into an R session
- To generate the basic package files, run `package.skeleton("packageName")`

## Step 2: Edit the package files

- Fill in the `DESCRIPTION` and help files (`man > .Rd`)
- Edit or add a `NAMESPACE` file
- Function or data updates should be done within the package files

## 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: Create the package files

## Process overview (the easy way)

- Load all data objects and functions to be included in the package into an R session
- Run the `package.skeleton` command with a single argument of the package name (in quotation marks) to generate the package files
- Learn where these package files got placed using the `getwd` function
- Find the package files in this folder and move them, if needed, to where you want the package files to live on your computer

## Alternative to the last two steps

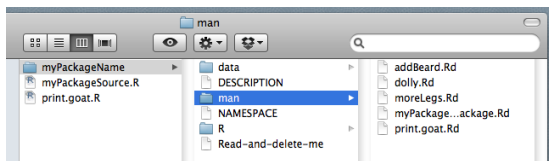
- An optional `path` argument is available in `package.skeleton` to specify a location where the package should be saved

## Step 1: Create the package files

```
> addBeard <- function(x){ x$beard <- TRUE;      return(x) }
> moreLegs <- function(x){ x$legs  <- x$legs+1; return(x) }
>
> dolly    <- data.frame(beard = FALSE, legs = 4, tails = 1)
> class(dolly) <- "goat"
>
> source("print.goat.R")
>
> package.skeleton("myPackageName")
Creating directories ...
Creating DESCRIPTION ...
Creating NAMESPACE ...
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: Create the package files

The package source folder, which has the same name as that specified in `package.skeleton`, contains several files and folders that were automatically generated



`data` (folder)

`DESCRIPTION`

`man` (folder)

`NAMESPACE`

`R` (folder)

`Read-and-delete-me`

Contains `.rda` files of each data object

General package information

Help files

Manages function, method, and dependency info

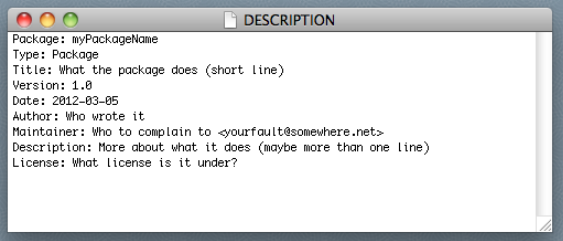
Contains `.R` files for each function

File to be deleted

## Step 2: Edit the package files – DESCRIPTION

### DESCRIPTION file instructions

- Update all information
- Choose your license (e.g. `GPL-3` or `GPL (>= 2)`)
- If the package is dependent on one or more other packages, create a new line in the `DESCRIPTION` file that starts as `Depends:` and list the required packages, separated by commas
- If the package depends on a later version of R, say version 2.10.1 or later, then this is accomplished by specifying `R (>= 2.10.1)` on the `Depends` line

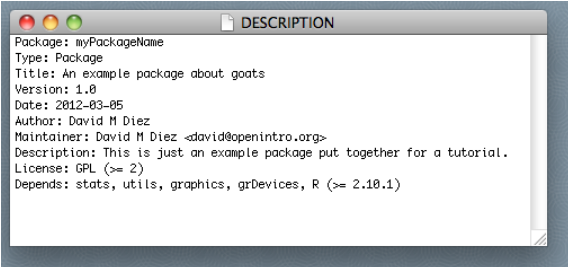


```
DESCRIPTION
Package: myPackageName
Type: Package
Title: What the package does (short line)
Version: 1.0
Date: 2012-03-05
Author: Who wrote it
Maintainer: Who to complain to <yourfault@somewhere.net>
Description: More about what it does (maybe more than one line)
License: What license is it under?
```

## Step 2: Edit the package files – DESCRIPTION

### Example of a revised DESCRIPTION file

- Notice the license option, which permits GPL version 2 *or later*
- The `stats`, `utils`, `graphics`, and `grDevices` packages are often already loaded in any R session, but it may be helpful to list them as dependencies
- The R version 2.10.1 dependency is listed in the `Depends` entry as an example, and it is not actually necessary for this package

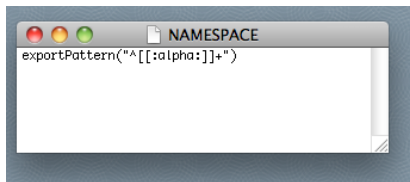


```
DESCRIPTION
Package: myPackageName
Type: Package
Title: An example package about goats
Version: 1.0
Date: 2012-03-05
Author: David M Diez
Maintainer: David M Diez <david@openintro.org>
Description: This is just an example package put together for a tutorial.
License: GPL (>= 2)
Depends: stats, utils, graphics, grDevices, R (>= 2.10.1)
```

## Step 2: Edit the package files – NAMESPACE

### Very basic NAMESPACE file

- Earlier versions of R don't automatically generate a NAMESPACE file, so add one if needed with no extension (eliminate the `txt` extension after the file is created, if it was added by the text editor)
- If there are no special functions you want hidden, no methods in the package, and no package dependencies, then just leave the file as-is
- If you had to make your own NAMESPACE (probably because you are using R version < 2.14.0), put in the `exportPattern` command listed below

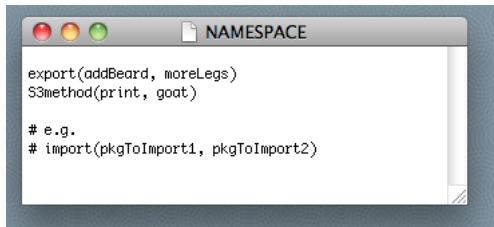


```
exportPattern("^[[:alpha:]]+")
```

## Step 2: Edit the package files – NAMESPACE

### Editing the NAMESPACE file

- To have hidden functions. Replace the `exportPattern` command with an `export` command, where `export`'s arguments are comma-separated function names that should be accessible to users
- Methods. To specify S3 method called `Method` for class "`Class`", create a line in the `NAMESPACE` file as `S3method(Method, Class)`
- Dependencies. Some users may prefer `Imports:` instead of `Depends:` in the `DESCRIPTION` file, and they must then also provide a command `import` in the `NAMESPACE` file whose arguments are the names of packages that the new package imports



```
export(addBeard, moreLegs)
S3method(print, goat)

# e.g.
# import(pkgToImport1, pkgToImport2)
```

## Step 2: Edit the package files – `man` files

### Basic rules of help (`.Rd`) files

- Notation is similar to  $\text{\LaTeX}$  where commands start with a backslash
- Use `\code{ }` to write in Courier
- Note: the `\example` section already uses Courier
- To create a link to a help file for a data object or function, say `addBeard`, use `\link{addBeard}`
- If the data object or function is from another package, its package name must also be in the link: `\link[otherPkg]{otherFcn}`
- Usually place `\link` command inside of `\code` but never vice-versa

### Equations with $\text{\LaTeX}$ notation require two new commands

- In-line equations use the `\eqn{ }` command instead of dollar signs
- Stand-alone one-line equations use `\deqn{ }`
- The  $\text{\LaTeX}$ -formatted equations will only show up in the package manual and otherwise appear plain



## Step 2: Edit the package files – `man` files

Delete help (`.Rd`) files for functions that are both not exported and not S3 methods

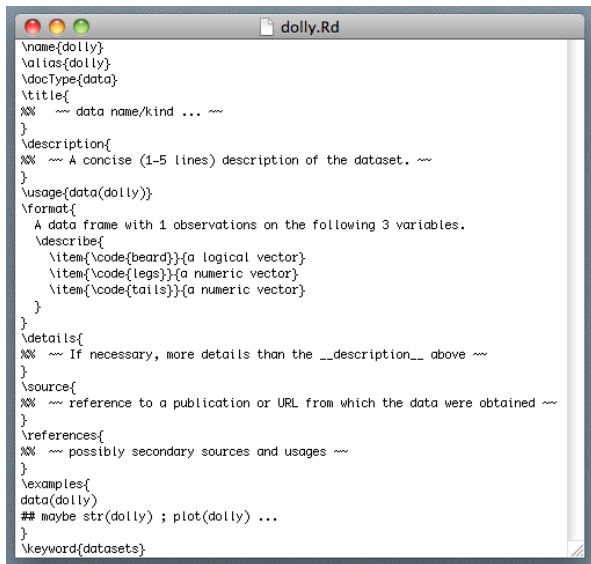
Follow the template instructions in each help file

- Must provide a title for every help file
- Delete sections that are not needed, perhaps `\details{ }` or `\references{ }`
- The package help file may have a few lines outside of any commands (starting with `~`), which should be deleted

Merging help files for two or more functions

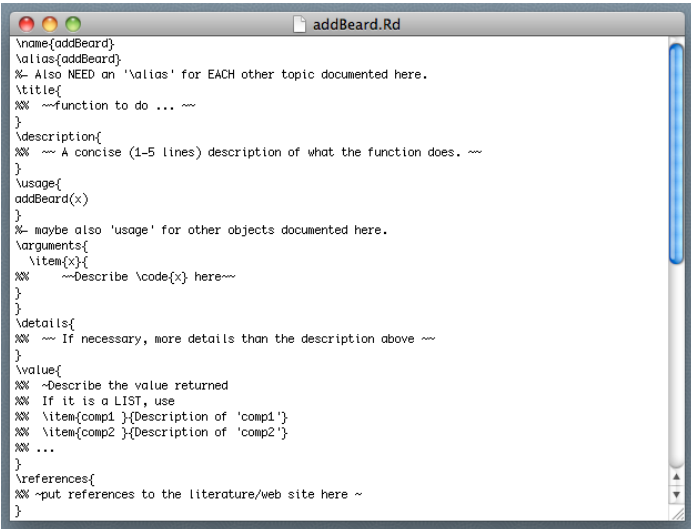
- Choose one help file that will be the help file for the functions
- Copy the `\alias{ }` and possibly also any `\usage{ }` commands from the other help files into this main help file
- Add in additional argument descriptions, as needed, and any supplemental descriptions to the merged help file
- Finally, delete the other help files

## Step 2: Edit the package files – `man` files (data)



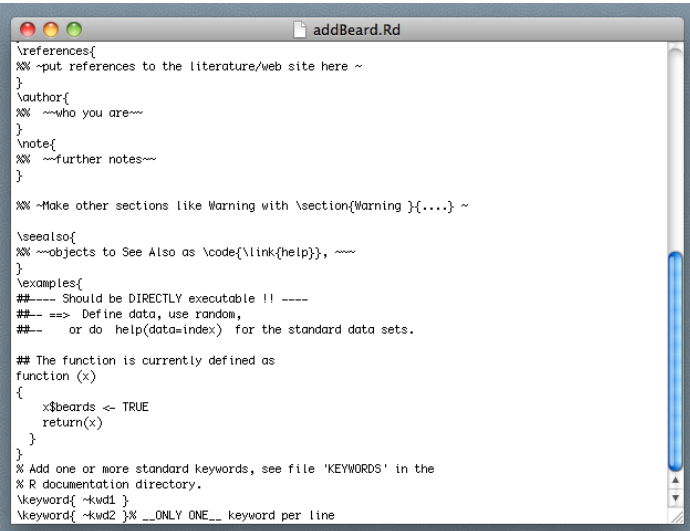
```
\name{dolly}
\alias{dolly}
\docType{data}
\title{
%% ~ data name/kind ... ~
}
\description{
%% ~ A concise (1-5 lines) description of the dataset. ~
}
\usage{data(dolly)}
\format{
  A data frame with 1 observations on the following 3 variables.
  \describe{
    \item{\code{beard}}{a logical vector}
    \item{\code{legs}}{a numeric vector}
    \item{\code{tails}}{a numeric vector}
  }
}
\details{
%% ~ If necessary, more details than the __description__ above ~
}
\source{
%% ~ reference to a publication or URL from which the data were obtained ~
}
\references{
%% ~ possibly secondary sources and usages ~
}
\examples{
data(dolly)
## maybe str(dolly) ; plot(dolly) ...
}
\keyword{datasets}
```

## Step 2: Edit the package files – `man` files (function)



```
\name{addBeard}
\alias{addBeard}
% 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{
addBeard(x)
}
% maybe also 'usage' for other objects documented here.
\arguments{
  \item{x}{
%% ~Describe \code{x} 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'}
%% ...
}
\references{
%% ~put references to the literature/web site here ~
}
```

## Step 2: Edit the package files – `man` files (function)



```
\references{
%% ~put references to the literature/web site here ~
}
\author{
%% ~who you are~
}
\note{
%% ~further notes~
}

%% ~Make other sections like Warning with \section{Warning}{...} ~

\seealso{
%% ~objects to See Also as \code{\link{help}}, ~
}
\examples{
##---- Should be DIRECTLY executable !! ----
##-- ==> Define data, use random,
##--   or do help(data=index) for the standard data sets.

## The function is currently defined as
function (x)
{
  x$beards <- TRUE
  return(x)
}
% Add one or more standard keywords, see file 'KEYWORDS' in the
% R documentation directory.
\keyword{ ~kwd1 }
\keyword{ ~kwd2 }% __ONLY ONE__ keyword per line
```

## Step 2: Edit the package files – adding data

To add a new data object, say object `obj`

- Load the data into R and save the data object to a file:  
`save(obj, file="obj.rda")`
- Create a help file: `prompt(obj)`
- See where the two files got saved: `getwd()`
- Move the files into the `data` and `man` folders, respectively
- If the package didn't already have a `data` folder, then add one

It is okay to replace an existing `.rda` file and then update the existing help file

## Step 2: Edit the package files – adding functions

Add a new function, say `fcn`

- Save the function declaration/definition to a `.R` file (a text file with a `.R` file extension)
- Load the function into R and generate a help file: `prompt(fcn)`
- See where the help file got saved: `getwd()`
- Move the `.R` file to the package's `R` folder and move the help file to the `man` folder

It is okay to update an existing `.R` file and then update the existing help file, but do be sure to update the usage and arguments sections, if needed

## Step 3: Build, check, and install on Mac OS X

### Quick-start directions

- Copy the package files to your [Desktop](#)
- Open Applications > Utilities > Terminal
- Navigate to the [Desktop](#) by typing

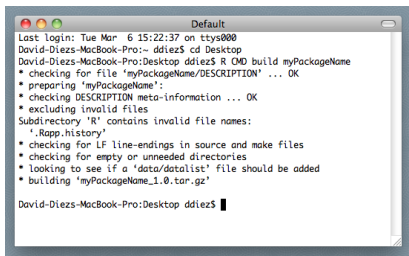
```
cd Desktop
ls
```
- The `ls` command above should print out a list of files and folders, one of which should be your package, but if not...
  - Type `pwd` and hit *return* to learn your present working directory
  - Copy the package files to this directory
- Users familiar with navigating in UNIX: feel free to modify the directions above as you see fit

## Step 3: Build, check, and install on Mac OS X

Inside of Terminal (iTerm is also okay)

- To build a `.tar.gz` file (a “tarball”) for sharing the package  
R CMD build myPackageName
- To install the package from its folder  
R CMD install myPackageName
- To check the package, perhaps before submitting to CRAN  
R CMD check myPackageName

Fix errors, often in package documentation, as needed



```
Default
Last login: Tue Mar  6 15:22:37 on ttys000
David-Diezs-MacBook-Pro:~ ddiez$ cd Desktop
David-Diezs-MacBook-Pro:Desktop ddiez$ R CMD build myPackageName
* checking for file 'myPackageName/DESCRIPTION' ... OK
* preparing 'myPackageName':
* checking DESCRIPTION meta-information ... OK
* excluding invalid files
Subdirectory 'R' contains invalid file names:
'.Rapp.history'
* checking for LF line-endings in source and make files
* checking for empty or unneeded directories
* looking to see if a 'data/datalist' file should be added
* building 'myPackageName_1.0.tar.gz'

David-Diezs-MacBook-Pro:Desktop ddiez$
```



## Step 3: Build, check, and install on Mac OS X

Problems with building, checking, or installing a package from UNIX may indicate that some software installations may be needed

- LaTeX compiler, such as the one in MacTeX, are generally required for checking a package  
[tug.org/mactex](http://tug.org/mactex)
- Apple Xcode, which is free for Lion but now difficult to come by for Snow Leopard, may be required on your computer  
[developer.apple.com/xcode](http://developer.apple.com/xcode)

## Step 3: The `check` step often requires attention

- 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 initially skipping the `check` stage
- CRAN will *not* accept a package that has warnings or errors from `check`

## Step 3 on Windows

Windows requires Rtools and, if running **check**, a LaTeX compiler

- **Required.** Install a copy of Rtools, which can be found on [cran.r-project.org/bin/windows/Rtools/](http://cran.r-project.org/bin/windows/Rtools/)
- Install a LaTeX compiler, such as [miktex.org](http://miktex.org)
- Users installing MikTeX may find the UCLA ATS website useful: [www.ats.ucla.edu/stat/latex/icu/install\\_win.htm](http://www.ats.ucla.edu/stat/latex/icu/install_win.htm)

The remaining details of packaging on Windows are not minor, so here's a good reference to keep you moving:

[stevemosher.wordpress.com/ten-steps-to-building-an-r-package-under-windows](http://stevemosher.wordpress.com/ten-steps-to-building-an-r-package-under-windows)

# Recap on package building mechanics

## 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
- Modify or add a `NAMESPACE` file
- 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

# 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

# Other random bits of knowledge

## Other files and folders in packages

- Reserved folder names: `demo`, `exec`, `inst`, `po`, `src`, `tests`
- The following file names are also reserved for special purposes: `cleanup`, `configure`, `INDEX`, `LICENSE`, `LICENCE`, and `NEWS`
- May add misc. files to `inst`, e.g. derivations, but keep total file size under about 5MB

## C, Fortran, etc in packages

- Guide to using C in R:  
[www.ats.ucla.edu/stat/r/library/interface.pdf](http://www.ats.ucla.edu/stat/r/library/interface.pdf)
- C, C++, and Fortran source code goes in the `src` folder
- Requires updates to `NAMESPACE` file
- Specify package name in the calls the C or Fortran code by specifying the `PACKAGE` argument in `.C`, `.Fortran`, etc (see `?C`):  
`PACKAGE="myPackageName"`

# 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](mailto: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` runs with no warnings or errors

## Uploading files

- Use an FTP client to upload files, such as Cyberduck (Mac)

## Keep in mind

- CRAN personnel 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](http://www.jstatsoft.org)
- R Journal – [journal.r-project.org](http://journal.r-project.org)

## Find the source of packages on their CRAN pages

```
stockPortfolio: Build stock models and analyze stock portfolios
Download stock data, build single index, construct correlation, and resample models, and estimate optimal stock portfolios. Plotting functions for the
portfolio possibilities curve and portfolio cloud are included. A function to test a portfolio on a data set is also provided.

Version: 1.1
Published: 2010-04-15
Author: David Diez and Nicolas Christou
Maintainer: David Diez <david.m.diez@gmail.com>
License: GPL-2
In view of: Finance
CRAN checks: stockPortfolio results

Downloads:
Package source: stockPortfolio\_1.1.tar.gz
Mac OS X binary: stockPortfolio\_1.1.dmg
Windows binary: stockPortfolio\_1.1.zip
Reference manual: stockPortfolio.pdf
Old sources: stockPortfolio.archive
```

# Helpful videos

## Screen-capture videos showing how to build a package on Mac OS X

[youtube.com/watch?v=d5TvxbtMZKg](https://youtube.com/watch?v=d5TvxbtMZKg)

[youtube.com/watch?v=TX5\\_6L991CQ](https://youtube.com/watch?v=TX5_6L991CQ)

[youtube.com/watch?v=qzCQHmPXax8](https://youtube.com/watch?v=qzCQHmPXax8)

## Rory Winston presenting on package building

[youtube.com/watch?v=8-dGf-7arFI](https://youtube.com/watch?v=8-dGf-7arFI)

# Helpful references

## Software for Data Analysis

[www.r-project.org/doc/bib/R-books.html](http://www.r-project.org/doc/bib/R-books.html)

John Chambers

Springer, 2008

## Creating R Packages: A Tutorial

[cran.r-project.org/doc/contrib/Leisch-CreatingPackages.pdf](http://cran.r-project.org/doc/contrib/Leisch-CreatingPackages.pdf)

Friedrich Leisch

Department of Statistics

Ludwig-Maximilians-Universität München

R Development Core Team

David M Diez  
david@openintro.org  
@RFunction (Twitter)  
RFunction.com  
ddiez.com