

# Building R Packages

*Roger D. Peng, Associate Professor of Biostatistics*

## What is an R Package?

- A mechanism for extending the basic functionality of R
  - A collection of R functions, or other (data) objects
  - Organized in a systematic fashion to provide a minimal amount of consistency
  - Written by users/developers everywhere
- 

## Where are These R Packages?

- Primarily available from CRAN and Bioconductor
- Also available from GitHub, Bitbucket, Gitorious, etc. (and elsewhere)
- Packages from CRAN/Bioconductor can be installed with `install.packages()`
- Packages from GitHub can be installed using `install_github()` from the devtools package

You do not have to put a package on a central repository, but doing so makes it easier for others to install your package.

---

## What's the Point?

- “Why not just make some code available?”
  - Documentation / vignettes
  - Centralized resources like CRAN
  - Minimal standards for reliability and robustness
  - Maintainability / extension
  - Interface definition / clear API
  - Users know that it will at least load properly
- 

## Package Development Process

- Write some code in an R script file (.R)
  - Want to make code available to others
  - Incorporate R script file into R package structure
  - Write documentation for user functions
  - Include some other material (examples, demos, datasets, tutorials)
  - Package it up!
-

## Package Development Process

- Submit package to CRAN or Bioconductor
  - Push source code repository to GitHub or other source code sharing web site
  - People find all kinds of problems with your code
  - Scenario #1: They tell you about those problems and expect you to fix it
  - Scenario #2: They fix the problem for you and show you the changes
  - You incorporate the changes and release a new version
- 

## R Package Essentials

- An R package is started by creating a directory with the name of the R package
  - A DESCRIPTION file which has info about the package
  - R code! (in the R/ sub-directory)
  - Documentation (in the man/ sub-directory)
  - NAMESPACE (optional, but do it)
  - Full requirements in Writing R Extensions
- 

## The DESCRIPTION File

- Package: Name of package (e.g. library(name))
  - Title: Full name of package
  - Description: Longer description of package in one sentence (usually)
  - Version: Version number (usually M.m-p format)
  - Author, Authors@R: Name of the original author(s)
  - Maintainer: Name + email of person who fixes problems
  - License: License for the source code
- 

## The DESCRIPTION File

These fields are optional but commonly used

- Depends: R packages that your package depends on
  - Suggests: Optional R packages that users may want to have installed
  - Date: Release date in YYYY-MM-DD format
  - URL: Package home page
  - Other fields can be added
- 

## DESCRIPTION File: `gpclip`

Package: `gpclip` Title: General Polygon Clipping Library for R Description: General polygon clipping routines for R based on Alan Murta's C library Version: 1.5-5 Author: Roger D. Peng [rpeng@jhsph.edu](mailto:rpeng@jhsph.edu) with contributions from Duncan Murdoch and Barry Rowlingson; GPC library by Alan Murta Maintainer: Roger D. Peng [rpeng@jhsph.edu](mailto:rpeng@jhsph.edu) License: file LICENSE Depends: R ( $\geq$  2.14.0), methods Imports: graphics Date: 2013-04-01 URL: <http://www.cs.man.ac.uk/~toby/gpc/>, <http://github.com/rdpeng/gpclip>

---

## R Code

- Copy R code into the R/ sub-directory
  - There can be any number of files in this directory
  - Usually separate out files into logical groups
  - Code for all functions should be included here and not anywhere else in the package
- 

## The NAMESPACE File

- Used to indicate which functions are exported
  - Exported functions can be called by the user and are considered the public API
  - Non-exported functions cannot be called directly by the user (but the code can be viewed)
  - Hides implementation details from users and makes a cleaner package interface
- 

## The NAMESPACE File

- You can also indicate what functions you import from other packages
  - This allows for your package to use other packages without making other packages visible to the user
  - Importing a function loads the package but does not attach it to the search list
- 

## The NAMESPACE File

Key directives - `export("<function>")` - `import("<package>")` - `importFrom("<package>", "<function>")`

Also important - `exportClasses("<class>")` - `exportMethods("<generic>")`

---

## NAMESPACE File: mvtsplot package

```
export("mvtsplot")
importFrom(graphics, "Axis")
import(splines)
```

---

## NAMESPACE File: gpclib package

```
export("read.polyfile", "write.polyfile")
importFrom(graphics, plot)
```

```
exportClasses("gpc.poly", "gpc.poly.nohole")

exportMethods("show", "get.bbox", "plot", "intersect", "union", "setdiff",
              "[", "append.poly", "scale.poly", "area.poly", "get.pts",
              "coerce", "tristrip", "triangulate")
```

---

## Documentation

- Documentation files (.Rd) placed in man/ sub-directory
  - Written in a specific markup language
  - Required for every exported function
  - Another reason to limit exported functions
  - You can document other things like concepts, package overview
- 

## Help File Example: line Function

```
\name{line}
\alias{line}
\alias{residuals.tukeyline}
\title{Robust Line Fitting}
\description{
  Fit a line robustly as recommended in \emph{Exploratory Data Analysis}.
}
```

---

## Help File Example: line Function

```
\usage{
line(x, y)
}
\arguments{
  \item{x, y}{the arguments can be any way of specifying x-y pairs. See
    \code{\link{xy.coords}}.}
}
```

---

## Help File Example: line Function

```
\details{
  Cases with missing values are omitted.

  Long vectors are not supported.
}
\value{
  An object of class \code{"tukeyline"}.
```

```
Methods are available for the generic functions \code{coef},  
\code{residuals}, \code{fitted}, and \code{print}.  
}
```

---

## Help File Example: line Function

```
\references{  
  Tukey, J. W. (1977).  
  \emph{Exploratory Data Analysis},  
  Reading Massachusetts: Addison-Wesley.  
}
```

---

## Building and Checking

- R CMD build is a command-line program that creates a package archive file (`.tar.gz`)
- R CMD check runs a battery of tests on the package
- You can run R CMD build or R CMD check from the command-line using a terminal or command-shell application
- You can also run them from R using the `system()` function

```
system("R CMD build newpackage")  
system("R CMD check newpackage")
```

---

## Checking

- R CMD check runs a battery tests
  - Documentation exists
  - Code can be loaded, no major coding problems or errors
  - Run examples in documentation
  - Check docs match code
  - All tests must pass to put package on CRAN
- 

## Getting Started

- The `package.skeleton()` function in the `utils` package creates a “skeleton” R package
  - Directory structure (`R/`, `man/`), `DESCRIPTION` file, `NAMESPACE` file, documentation files
  - If there are functions visible in your workspace, it writes R code files to the `R/` directory
  - Documentation stubs are created in `man/`
  - You need to fill in the rest!
-

## Summary

- R packages provide a systematic way to make R code available to others
  - Standards ensure that packages have a minimal amount of documentation and robustness
  - Obtained from CRAN, Bioconductor, Github, etc.
- 

## Summary

- Create a new directory with R/ and man/ sub-directories (or just use `package.skeleton()`)
- Write a DESCRIPTION file
- Copy R code into the R/ sub-directory
- Write documentation files in man/ sub-directory
- Write a NAMESPACE file with exports/imports
- Build and check