R Package Creation

Creating Well Documented and Reusable Software

Spencer Lourens

April 25 2017

Outline of Lecture

- What is an R package?
- Why use R packages?
- Creating R packages in RStudio - The easy way
- Package Structure
- Documentation
- Shipping data, tests
- Vignettes
- Simple workflow - devtools cheatsheet
- A minimal example package utilizing S3 and S4 OOP systems

What is an R Package?

- "A package is the fundamental unit of shareable code in the R language." - Hadley Wickham
  - Code
  - Metadata / Namespace info
  - Data
  - Documentation
  - Tests/Examples
- Thousands of packages that extend R functionality can be found on **CRAN**, **Bioconductor**, and **github**
- An R package is essentially a main directory with subdirectories that are organized in a very specific manner

Why Use an R Package?

- R packages make it easy to share your code with others
  - Sharing your code with others helps make it better
    - Find bugs, inefficiencies in computation or software set-up
    - Just as writers have other writers read their work, so should developers
    - We all get too close to our own work, and another set of eyes or experiences can help improve
- Better yet, R packages make it very easy to reuse your own code
  - No need to rewrite that pesky generalized linear hypothesis test (glht()) code again!
  - Summarize modeling results in a simple table format - simplify an RMarkdown workflow
  - Save yourself time, standardize your output!
- Deliver your work to the world! (PhD Dissertations)

Creating a package in RStudio

- RStudio has made this process much easier via the use of projects (or the devtools package)
- To create a package in RStudio:
  - File -> New Project
  - Choose "New Directory"
  - Choose "R Package"
  - Name your package, then click "Create Project"
  - Actually, you should make use of projects as much as possible, not just for packages - it confines your workspace and assures reproducibilty across systems
- Can also do this using the devtools package
  - devtools::create("/complete/package/path/package_name")
- Used to use package.skeleton() or create from scratch using the "Writing R Extensions" manual
- Don't use package.skeleton(), this will create way too much extra work for you in the end

Creating a package in RStudio

- You now have the basic set-up for an R package in the location you specified in the last slide
- The following components exist here:
  - A folder with name equal to the package name
  - An R/ directory
  - A NAMESPACE file
  - A DESCRIPTION file
- Each of these components has a specific use/meaning in an R package

Package Structure

- A package is just a directory organized into subdirectories
- Main directory name is the name of your package, i.e. SpenceRPackage
- R/
  - All R programs go here - no folders/directories!
- man/
  - All R documentation for R functions go here
- NAMESPACE file
  - Where does your package look for a function? (not the best, better to always use namespace::function())
  - What functions in your own package do you want users to have access to? (exports)
- DESCRIPTION file
  - Who wrote the package, what does it do, . . .
  - What packages does the package "depend" on, import, suggest - in general, don't use depends

Package Structure - R/

- All of your R programs reside here
- If your package has several functions - use multiple .R files
- Give them meaningful names
  - utility_funcs.R
  - plotting_funcs.R
  - modeling_funcs.R
- Assume case insensitivity (some operating systems are case-insensitie)
- Hadley's rule: if I can't remember where a function lives, I need more files, or better names for files

Package Structure - DESCRIPTION file

- Stores important metatadata for the package
- Includes package title, versions, authors (who to contact if things go wrong!), license, description
  - Depends: what is NEEDED to run your package (but it will load/attach, modifying the search path.. don't do this unless for R version)
  - Imports: what is NEEDED to run your package
  - Suggests: what is SUGGESTED to run your package
  - Packages specified in imports will be installed with your package automatically
- License is important to understand when you want to released your package to others, but we won't delve into that here

Package Structure - DESCRIPTION file

- Imports:
  - Include version of package
  - Safest to choose version you use and above
  - When using functions from other packages, use package::func()
  - Imports only makes the namespace accesible, it doesn't call library()
- Suggests:
  - You might use suggests if the main functionality in your package doesn't require the specified package
  - Packages in suggests are NOT automatically installed with your package
  - You will need to send a message to the user if they attempt to use functionality from a package in Suggests and do not have the package installed
  - isNamespaceLoaded("pkg", quietly = TRUE)

Package Structure - NAMESPACE file

- Two components: imports and exports
- Imports:
  - Determines how your package finds a function. In which package does it look?
  - This helps tremendously when there are multiple packages loaded, sometimes with the same name
  - Hmisc::summarize(), dplyr::summarize()
  - Best practice is to Imports: in description, call namespace::function() for future you
- Exports:
  - Determines which functions are available for use outside of your package
  - Some functions are only needed for internal use
  - The optimization routine or functions responsible for calculating the score/Hessian may not be needed by a user of your package
  - You may want users to be able to access the log-likelihood, residuals, etc.
  - Export as few functions as necessary to avoid conflicts with existing packages

Package Structure - NAMESPACE file

- Depends vs Imports
  - Depends loads and attaches a package
  - Imports only loads a package
- Loading a package loads all data, code, DLLs, etc. from a package
  - Keeps in memory: you must use package::func() to access a function
- After loading a package, you can attach that package
- This adds the package to the search path of R, so you can simply use func() without the package:: (namespace)
- Preferable to just put packages in imports, not depends
  - This means that you will always use package::func() in your package
  - Minimizes changes to the global environment - GOOD!
  - Interfere with user system as little as possible
  - I ALWAYS use namespace::function() to call functions in a package, i.e. I use imports not depends

Developing an R package in RStudio

- Use RStudio projects makes package development much easier
  - When you have a project open, only that project workspace is modified - isolation
  - We really should make use of projects in all of our workflow, be that collaboration, RA work, reserach work, etc. because it creates an isolated workspace, and you can send project folders to collaborators or put them on github
  - This means you never use a global search path, but instead relative search path which will start at the project directory level when you open the Rproject (easier for collaboration)
  - Keep data in project directory, write plots/reports to project directory, etc.
  - Navigate to function with ctrl + ., F2
  - You can test how your package would actually work when installed/loaded/attached
  - You can also do this with devtools::create(), devtools::check(), devtools::build, devtools::install()

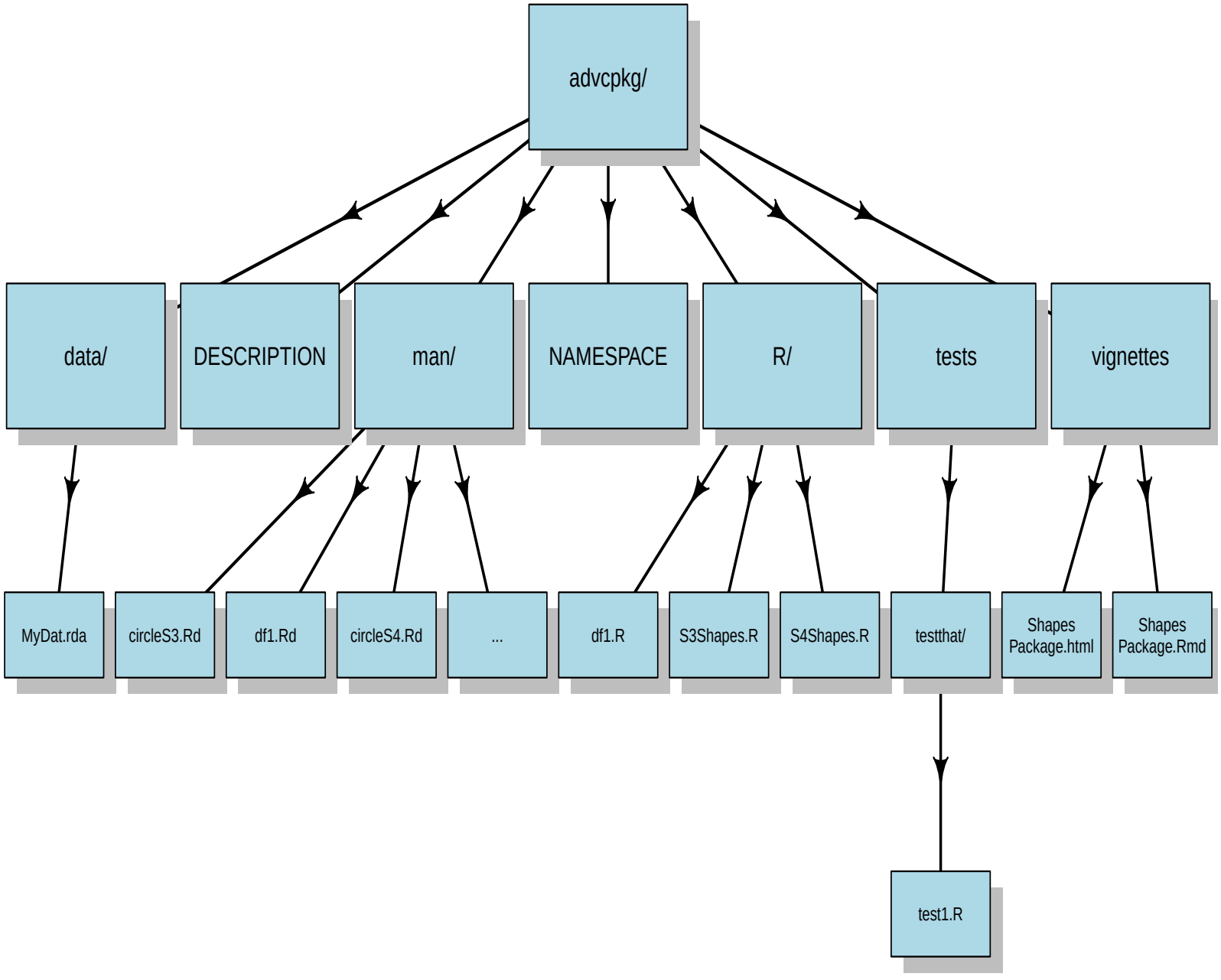
Documenting your package

- Package documentation is a very important step in package development
  - Tells users how package works
  - Default values in functions, context/examples
- The roxygen2 package (from Hadley, of course) makes documenting functionality from your package much easier
- If you are using S4 or other more complicated OOP structures/C++ code, you will have to look into other roxygen2 paradigms.. make StackOverflow your friend. There is NO WAY that this can all be covered in one lecture
- R package developers used to have to document our functions manually by reading "Writing R Extensions", and messing up a lot

Shipping datasets with your package

- You may want to ship data with your package
  - Demonstrate functionality in a vignette, or function help page
  - Datasets go in the /data directory, which should go in the main package directory
  - Datasets are also documented in an R package - you'll see this in the example package at the end
  - I suggest reading R Packages - Data for more information

Flowchart of overall package structure



Testing your package

- Use the testthat R package
- Tests you create go in the /tests subdirectory within the main package directory
- Tests assure that when subsequent changes to your package break previous functionality, you can easily see where the problem arose
- When you design your software, extra work up front to compartmentalize it in a sensible manner, i.e. one function for log likelihood calculation, one function for gradient calculation, one function for hessian calculation, etc.
- Allows creation of tests that can alert when created a bug in your code, have a higher chance of knowing where
- Make sure your newest version isn't a disaster, and users will stay loyal
- Three short tests provided in the example package - tests are NOT required to check a package, accomplish via devtools::test()

Vignettes

- A vignette is similar to an academic paper explaining your package
- Help pages are only so useful.. have to know what function you are looking for
- Vignette allows you to explain the idea behind the package, how you expect it to be used
- Vignettes (or portions of vignettes) can also be published within methodological papers, or as papers themselves (JCGS, JSS, . . .)
- Hadley refers to Vignettes as "Long Form Documentation" in his book on R Packages, which I strongly recommend - much more accessible than Writing R Extensions and free online
- I've compiled a short vignette for the package that I'll demonstrate - very simple. In your packages, you'll want to explain how to use it with an actual dataset (that may be simulated)
- Vignettes can be transformed into journal articles, so it's not a waste of time

Devtools Workflow

- See the devtools cheatsheet
- The basic workflow you can use is:
  - Update R code, roxygen documentation blocks
  - (Optional) run devtools::document(packageFilePath)
    - Converts roxygen comments to .Rd files, places in man/
  - if you want to run your unit tests: devtools::test(packageFilePath)
  - run devtools::check(packageFilePath) to test if any errors/warnings or useful notes are flagged
    - in fact, check() also updates documentation, builds, checks the package, and loads/attaches so you can use it
  - (Optional) run devtools::build(packageFilePath) to build the package as a bundle (tarball .tar.gz) which
- Your workflow may change from time to time based on your requirements, for instance you may not be in the testing stage yet but are trying to work out kinks from roxygen2

Using R's OOP (object oriented programming) Systems

- R has OOP systems that you can read about in Hadley's Advanced R book: Advanced R
- These OOP systems help you to change general R functions, called generics, such as print, plot, predict, etc. to behave in a certain way when objects from your own R package are used as arguments
- I'll next demonstrate an R package demonstrating the S3 and S4 systems using the generic shape class, and classes that inherit from shape: triangle, circle, and square
- We'll see that different characteristics and functions apply based on the class of the shape

Resources for Success

- Check out the R Packages book from Hadley Wickham, free online: R Packages
- Check out the Advanced R book from Hadley Wickham, also free online: Advanced R
- When necessary, brave your way through the original Writing R Extensions manual: Writing R Extensions

An Example R Package

- On github, the PackageCreation directory has another directory, advcpkg, within it
- This is an example R package that we'll now briefly go through - hopefully it is helpful when you document, test, and create vignettes for your own R packages
- This package only demonstrates S3, S4, but not reference classes (I think there are actually other OOP systems in R, which are less developed and thus more esoteric)
- Documenting S3 methods, S4 methods is a true PAIN. Hopefully this package will make it a lot easier for you since I have put a lot of hours into getting it just right - sadly, it will probably all change some time down the road
- In order to make it easier, it's all done through roxygen2 so you won't have to manually write documentation pages