Building Packages in R

David Diez
RFunction.com, OpenIntro

February 9, 2013

The success of R statistical software is in large part due to its flexible structure that allows researchers to develop supplemental software packages. These packages are commonly used to share data, implement new methods and tools, introduce students to statistics and computation, and support open research. This paper serves as a tutorial for individuals to create their own packages and release those packages on the Comprehensive R Archive Network (CRAN). Several topics are discussed: coding practices in R, preliminary steps in constructing a package, building S3 classes and methods in R, and completing an R package. This tutorial is supplemented by online videos and open source lecture slides.

Keywords: R, package building, tutorial

1 Introduction

The ease in generating and sharing code *packages* has helped R become an extremely successful software. This tutorial provides information necessary for beginner and intermediate R programmers to develop high-quality packages for internal use or for distribution. Packaging code provides immediate and long term benefits. To start, building an R package can facilitate collaboration, good programming practices, and help in the organization and documentation of code and data. In the long term, it provides support to researchers who must revisit their data or code, perhaps based on reviewer comments, or who seek to share data or software with other researchers.

It is assumed that readers are familiar with basic data objects such as vectors, matrices, data frames, and lists. It is also assumed that readers can comfortably create R functions with multiple arguments and have passing familiarity with the structure of R documentation (help) files accessed via the `?` operator or `help` function. Readers unfamiliar with these topics may find Venables et al. (2012) and Dalgaard (2008) useful. As this is a tutorial and users may be unfamiliar with many of the more complex functions, most functions have been linked in the PDF version to a blog (*RFunction.com*) that features clear and concise guidance for these functions.

The next four sections of this tutorial are organized into three instructional sections followed by a discussion. Package planning and helpful coding practices are introduced in Section 2. The construction of S3 classes and methods in R, useful for streamlining the experience of the end user, are discussed in Section 3. Section 4 covers the mechanics for creating an R package from existing functions and data. Lastly, the CRAN submission process and additional references are discussed in Section 5. There is little interdependency among the sections, and **readers may skip to those topics of greatest interest**. Open source lecture slides have also been made available to facilitate instruction on the topics presented in this paper (Diez, 2012), and screen-capture videos have been made available at RFunction.com[1] for a complete demonstration of the package building steps covered in Section 4.

2 Responsible programming practices

Successful packages have clear objectives, are easy to use, and are reliable. This section provides guidance on achieving each of these goals through planning and testing.

2.1 Preliminary steps in constructing a package

Packages are constructed to meet objectives, and it is helpful to understand explicitly what those objectives are prior to creating the package. Typically these objectives include a mission, an audience, and a scope. The mission declares the purpose of a package, such as whether the goal is to make available a new statistical method. A package creator should also consider the knowledge and experience level of the targeted end-user. Lastly, the scope of the project should be considered and established at an early stage to help reduce the temptation to put off a package's release to add one more function, which can be repeated indefinitely.

Why is this package important, necessary, and timely? Answering this question shapes the package contents. For instance, if a specific type of object will become common in the package, it is a good idea to create a class with methods for that type of object very early in the project (see Section 3). The answer to this question may also help in answering the next two questions.

Who is the target audience? The type of end-user may impact the interactive form of the package. This is perhaps most evident in packages aimed at introductory-level statistics courses, such as `Rcmdr` (Fox, 2005) and `Deducer` (Fellows, 2011).

What functions, data, and methods will be in the package? If this list is quite long, then also answer a second question: which of these should be available in a first release? These questions indicate what a final package might look like, possibly after multiple releases, and provide a perspective of what is sufficient for the package to be useful to others. It is tempting to postpone the release of a package because new features are being constantly added. However, it is important to remember that, unlike papers, packages may be updated.

Answering these three questions at the outset of a project will clarify the path for achieving aims and completing the project in a reasonable timeframe.

[1] http://rfunction.com/blog/building-packages

2.2 Coding implementation

Suppose a team of programmers, end-users, and scientists were tasked with the construction of a scale that rated the quality and effectiveness of R code. The specifics may differ, but they are likely to be represented by four aims: the code must produce correct results, be efficient, reviewable, and have a satisfactory user-interface. We do not suppose this is a complete list of all characteristics that are important to coding. Rather, these are items likely to be universally accepted (even in this regard, the list may be incomplete). This section highlights how one might achieve each the proposed four aims.

In addition to the systematic and specific comments listed below, it is appropriate to reiterate some time-tested advice: have a fellow programmer review and comment on your code and coding practices.

2.2.1 Correct

Accurate results are the most important standard for code. Software should be tested before any analysis results or software versions are released. One helpful approach to evaluating correctness is to test code in a systematic manner, which can be accomplished in three steps, repeated as necessary.

1. Data inputs are places where substantial diversity should be anticipated in a function. Create a list of ways a user's data may vary. For example, the measurement scale of data is one easily-overlooked element, as are violations of assumptions or the memory size required for the data set.

2. Simulate new data that varies widely across each of the characteristics identified in 1. Specify at least 2 tests per characteristic and record expectations for the results prior to applying code.

3. Apply the code to each of the test cases, and investigate cases where expectations do not match results. Fix bugs in the code that arise due to variations in data structures that were not anticipated during the construction of the code.

The list and test cases may be incomplete, and it may be appropriate to append new items and test cases as necessary. There are several tools available in R for debugging or testing. Many of these resources are discussed by Peng (2002). A few of the helpful debugging functions are provided in the footnote.[2] In addition to this methodological approach, there remain other common errors made in R that may be overlooked; two examples are provided in the footnote.[3]

[2] The `debug` function is used to put a function into a debugging mode, where the function can be reviewed in progress (see also: `undebug`). The `traceback` function traces an error through layers of functions to its source. That is, it will show the actual location of an error and how it was nested within other functions. Another function called `browser` is useful for browsing variables at a certain point within a function call. For instance, if the `browser()` command is added at line 20 of a function, then when the function is called it will go into the *browser* mode and allow variables in the function's environment to be queried. This is similar to `debug` except that it is no longer a one-step-at-a-time walk-through of a function but rather a single location within the function to investigate.

[3] 1. Using `T` and `F` for `TRUE` and `FALSE`. If a user over-writes the `T` or `F` variables (e.g. perhaps specifying `T=0` for time zero), then R will use those global variables, not `TRUE` and `FALSE`. 2. When calling a function recursively, use `Recall` instead of the function's name. This reduces the chance of errors when function names are later changed.

2.2.2 Efficient

Efficient code increases productivity and reduces effort, expense, and wait time of the end-user. Two goals are considered in this section: curtail the creation of large data objects in a function that may stall computers with limited memory, and reduce run time. A systematic approach for addressing the limited memory and run time goals for each of the large or complex functions in a software release is described below:

1. Identify 3-5 test cases, where at least two include complex and big data, where *complex* and *big* are well-defined only within the context of the problem at hand. These test cases may already have been developed for checking the code's correctness.

2. Add the following warning at the top of the function to be tested:

   ```
   warning("Function is currently in testing mode\n") # ZZQ
   ```

   This is a reminder that this function must be revised again to remove the tools that will be inserted. The commented string of characters (`ZZQ`) is added to each line of code that needs elimination or revision following testing.

3. If there are concerns about memory use in a function, choose 1-2 large objects created in the function. At appropriate stages in the function, print the size of these objects by using the `cat` and `object.size` functions:

   ```
   cat("objName", object.size(objName), "\n") # ZZQ
   ```

   In this example, `objName` is the name of the object under investigation.

4. Identify blocks of code that are to be timed separately. At the start of the function, initialize time and block variables:

   ```
   Time  <- Sys.time() # ZZQ
   Block <- c() # ZZQ
   ```

   The `Sys.time` function records the present time on the computer. At the end of each coding block, append a new time and describe the code in the block:

   ```
   Time  <- append(Time, Sys.time()) # ZZQ
   Block <- append(Block, "What this block time corresponds to") # ZZQ
   ```

   Finally, return the `Time` and `Block` variables upon the completion of this function.

   ```
   return(list(Time=Time, Block=Block)) # ZZQ
   # return(objectOriginallyReturned) # ZZQ
   ```

   The time difference for the blocks may be obtained via `diff(out$Time)`, where `out` is the returned object.

5. Run the test cases with the modified function.

Use the information returned to determine whether and how the function should be revised. It is often useful to initially devote time to the largest objects and slowest-running blocks. A few additional tools and comments regarding efficiency are provided in the footnote.[4]

[4]The `Rprof` function tries to achieve the aims of the systematic approach described and it may be helpful to some users, but often its recorded information is inadequate for a careful review. The `system.time` function is helpful

2.2.3 Reviewable

For code to be reviewable, a programmer given the code and documentation should be able to follow the general flow of the code. Three general programming rules are provided in the footnote,[5] but otherwise the tips below attempt to make code easier to read and review:

- Visually partition code into variables and the commands used to generate those variables by aligning assignment characters within blocks of code. This typically requires extra white spaces to the left of some assignments.

```
id2     <- c(1, 2, 6, 7, 8, 9, 10)
weights <- c(147, 113, 168, 135, 144, 173, 128)
d2      <- data.frame(id=id2, weights)

beta  <- c(2.1, 6.21, 4.321, 5.4321)
SE    <- c(5, 1.49, 11.13, 0.31)
t     <- beta/SE
```

  Alignment facilitates the exploration and review of variables and their origins.

- Provide brief, targeted comments in the code. Reserve long explanations for a function's documentation, such as the *Details* section in its help file.

- Place comments within a set of characters that create a horizontal break, such as one of the following three examples:

```
#______ Brief, Targeted Comment ______#

#------ Brief, Targeted Comment ------#

#====== Brief, Targeted Comment ======#
```

  The horizontal-orientation of _, -, and = draw attention during vertical scrolling.

- Do not use comments to explain what a function does, which is the purpose of help documentation. (It's never too early to create an R packge.)

for checking the speed of a single line of code. Quick tips: (1) The apply function should be used in place of loops whenever possible. (2) Initialize an entire vector when possible rather than growing the vector in a loop. (3) Do not recompute values unnecessarily in a loop.

[5] (1) Develop functions for processing and analyzing data. (2) Use indentation to show code structure, e.g. to indicate the contents of a loop or function. (3) Create line breaks to section code into blocks.

2.2.4 User-friendly

Software must meet some baseline of usability, and in R, standards for user-friendliness are high. Here, three topics are considered: documentation, nomenclature, and feedback.

Documentation. Documentation must be clear and concise with easy-to-follow examples. When these aims cannot be achieved, it is symptomatic of deeper issues with how the functions or software were constructed. If necessary, revise functions. Complex data processing should be contained within functions whenever reasonable.

Nomenclature. If an argument in a new function is analogous to an argument in a commonly used function, use that same argument name to reduce the amount of new nomenclature. Choose object names that indicate their content. Create classes and methods for complex or large objects to encourage the use of familiar functions.

Feedback. Ensure all functions return sensible warnings and errors, when appropriate. The top of every user-friendly function contains several input checks. If an input does not satisfy basic checks, use `stop` and `warning` to display an error or warning that clearly indicates the problem and, if it can be done briefly, how to fix the problem.

There are other techniques that can improve the experience of a user in R. For example, create a progress bar for a slow function using `txtProgressBar` and `setTxtProgressBar`, or return invisible information from a plotting function by using `invisible` instead of `return`.

3 S3 class construction and methods

Classes and methods are used to smooth the user experience when handling complex objects that are returned from functions. A **class** is a set of objects that share specific attributes and a common label. A **method** is a name for a function or action that can be applied to many types of objects. Objects need not be very complex to reap benefits from the class/method structure, though classes and methods become increasingly beneficial as complexity increases.

Each object in R has a class, which generally indicates the type of information contained in the object. Some common classes are `"numeric"`, `"matrix"`, `"list"`, and `"lm"`. This brief list might suggest that an object's class is always the same as the function used to generate the object; this structure is not a rule.

We will see that the class of an object – in the S3 sense – can be *assigned* to an object, and this is generally done within the function that generates the object. We will also see that object classes integrate with methods. For instance, `print`, `summary`, and `plot` are well-known functions that are actually methods, and they can handle many classes of objects. Common method functions may be viewed as something like a phone operator; methods first recognize an object's class then pick an appropriate function to apply to the object. For instance, the body of the print function is shown below:

```
> print
function (x, ...)
UseMethod("print")
```

```
<bytecode: 0x117dbb468>
<environment: namespace:base>
```

There isn't much to the function: it simply redirects the `x` object and any additional arguments to the `print` method corresponding to the class of `x`.

3.1 Becoming familiar with S3 classes and methods

R supports two class structures called S3 and S4, and we will focus on the simpler S3 classes, which are easy to both dissect and construct.

In R, an object's class may be accessed using the `class` function:

```
> x <- 1:10
> y <- 5 + 0.2*x + rnorm(10)
> class(x)
[1] "integer"
> class(y)
[1] "numeric"
> g <- lm(y ~ x)
> class(g)
[1] "lm"
```

Knowing an object's class is important for three reasons. First, it often (but not always) implies how the object was generated. Second, it suggests the object's structure. For example, if we encounter an object of class `lm`, we would expect it to be a list with a particular set of items, such as `coefficients` and `residuals`, which summarize the model. Third, methods behave differently depending on the class of their primary argument. For example, the `plot` method automatically creates something sensible for both `numeric` and `lm` objects. Furthermore, methods for some classes will include additional hidden arguments for further customization.

Complex S3 objects in R are often initially built as a list, then the class of the object is changed. For instance, the `lm` object is actually a list but with the class modified to be `"lm"` rather than `"list"`. We could examine further details of an `"lm"` object – and many more S3 class objects – using the `names` function, `$` operator, or `str` function.

```
> #====== g is an lm object from above ======#
> class(g)
[1] "lm"
> names(g)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
> g$residuals
          1           2           3           4           5           6
 1.17393095 -0.31214680 -0.43396725 -0.04293141 -0.69272489 -0.45055872
          7           8           9          10
-0.32780308  0.85521155 -0.23399951  0.46498916
```

Generally the S3 class of an R object may be modified using the `class` function. To generate a new class of objects, simply assign a new class name.

```
> #====== Make g into simple list ======#
> class(g)
[1] "lm"
>
> class(g) <- "list"
> summary(g)
              Length Class      Mode
coefficients   2     -none-     numeric
residuals     10     -none-     numeric
effects       10     -none-     numeric
rank           1     -none-     numeric
fitted.values 10     -none-     numeric
assign         2     -none-     numeric
qr             5     qr         list
df.residual    1     -none-     numeric
xlevels        0     -none-     list
call           2     -none-     call
terms          3     terms      call
model          2     data.frame list
>
> class(g) <- "myClass"
> class(g)
[1] "myClass"
```

S3 classes have an incredible potential for the regular R software writer to create a rich, clean user experience. The rub is that methods must be written for the new class. Thankfully this task is also straightforward.

3.2 Writing methods for an S3 class

This section outlines the creation of the `print` method for an object of class `"stockReturns"` from a function called `getReturns` in the `stockPortfolio` package (Diez and Christou, 2012). Three of the most common methods are `print`, `summary`, and `plot`. Others, such as `predict` and `residuals`, may also be especially useful for software that implements a new statistical technique.

To write a `print` method for the `"stockReturns"` class, a new function called `print.stockReturns` is created. In general, to generate a method `method` for an object of class `"newClass"`, a function called `method.newClass` is written. Therefore, the `print` method for the `"stockReturns"` class is generated by creating a function of the following form:

```
print.stockReturns <- function(x, ...){
  # put code to print a stockReturns object here
}
```

The first argument, `x`, is standard and must be maintained for the `print` method. Other methods have alternative argument names, e.g. `object` for the `summary` method.

Before writing the `print.stockReturns` function, it is important to understand the structure of an object of class `"stockReturns"`. The function `getReturns` retrieves specific stock data from Yahoo! Finance, preprocesses the data, and returns a list with class `"stockReturns"`. Below, stock data since January 2008 for Apple, Microsoft, and Red Hat is retrieved using `getReturns`, where the result has been stored in `gr`. Next, the class and structure of `gr` are queried using the `class` and `str` functions.

```
> library(stockPortfolio)
> stocks <- c("AAPL", "MSFT", "RHT")
> gr     <- getReturns(stocks, start="2008-01-01")
> class(gr)
[1] "stockReturns"
> str(gr)
List of 5
 $ R     : num [1:49, 1:3] 0.1518 0.1271 0.0597 -0.0558 0.0615 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:49] "2012-02-01" "2012-01-03" "2011-12-01" ...
  .. ..$ : chr [1:3] "AAPL" "MSFT" "RHT"
 $ ticker: chr [1:3] "AAPL" "MSFT" "RHT"
 $ period: chr "month"
 $ start : chr "2008-02-01"
 $ end   : chr "2012-03-01"
 $ full  :List of 3
  ..$ AAPL:'data.frame': 51 obs. of  7 variables:
  .. ..$ Date     : Factor w/ 51 levels "2008-01-02","2008-02-01",..:
51 50 49 48 47 46 45 44 43 42 ...
  .. ..$ Open     : num [1:51] 548 458 409 383 397 ...
  .. ..$ High     : num [1:51] 610 548 458 409 408 ...
  .. ..$ Low      : num [1:51] 516 454 409 378 363 ...
  .. ..$ Close    : num [1:51] 596 542 456 405 382 ...
  .. ..$ Volume   : int [1:51] 26884200 21975800 12924800 11030900
15936300 23004900 21301100 25877600 20155000 15563200 ...
  .. ..$ Adj.Close: num [1:51] 596 542 456 405 382 ...
  ..$ MSFT:'data.frame': 51 obs. of  7 variables:
  .. ..$ Date     : Factor w/ 51 levels "2008-01-02","2008-02-01",..:
51 50 49 48 47 46 45 44 43 42 ...
  .. ..$ Open     : num [1:51] 31.9 29.8 26.6 25.6 26.2 ...
  .. ..$ High     : num [1:51] 33 32 29.9 26.2 27.2 ...
  .. ..$ Low      : num [1:51] 31.5 29.7 26.4 25.2 24.3 ...
  .. ..$ Close    : num [1:51] 32 31.7 29.5 26 25.6 ...
  .. ..$ Volume   : int [1:51] 46771300 52182700 70271500 49264800
53693200 60235300 63522800 77332100 68186700 61377000 ...
  .. ..$ Adj.Close: num [1:51] 32 31.7 29.3 25.8 25.4 ...
  ..$ RHT :'data.frame': 51 obs. of  7 variables:
```

```
  .. ..$ Date     : Factor w/ 51 levels "2008-01-02","2008-02-01",..:
51 50 49 48 47 46 45 44 43 42 ...
  .. ..$ Open     : num [1:51] 49.7 46.9 42.2 49.9 47.7 ...
  .. ..$ High     : num [1:51] 52.6 50.6 48.5 53 53.4 ...
  .. ..$ Low      : num [1:51] 48.2 46.2 41.3 39.2 44.9 ...
  .. ..$ Close    : num [1:51] 51.9 49.5 46.4 41.3 50.1 ...
  .. ..$ Volume   : int [1:51] 1467700 1286700 2086500 3405700 1945300
2732200 3196600 3816700 2129700 2590300 ...
  .. ..$ Adj.Close: num [1:51] 51.9 49.5 46.4 41.3 50.1 ...
 - attr(*, "class")= chr "stockReturns"
```

The object `gr` is a list with five items. The first item is a numerical matrix with 49 rows and 3 columns; further investigation would show that these are the returns for 49 months. The second item, `ticker`, lists the three tickers for Apple, Microsoft, and Red Hat. The third, fourth, and fifth items are the period for the returns (monthly), and the start and end dates. The last element in the list is actually another list with three items, where each item is the raw data from Yahoo! Finance for one stock.

Here, two options for the `print` method of the `"stockReturns"` class are considered.

1. Simply print out everything in the list, less the raw data. This may be appropriate since only one of the remaining list items is lengthy. Below the method is created and applied, where some output has been omitted.

```
> print.stockReturns <- function(x, ...){
+   x$full <- NULL
+   print.default(x) # prints it like a normal list
+ }
> gr
$R
                    AAPL         MSFT           RHT
2012-03-01  0.098831207  0.008506616  0.0487262434
2012-02-01  0.188310550  0.081799591  0.0666379124

...

$end
[1] "2012-03-01"

attr(,"class")
[1] "stockReturns"
```

2. Print out a short summary of the returns over the time range, where the returns are given relative to the time scale.

```
> print.stockReturns <- function(x, ...){
+   totalReturn <- apply(1 + x$R, 2, prod)
+   aveReturn   <- totalReturn^(1/nrow(x$R)) - 1
+   cat("Time Scale:", x$period, "\n\nAverage Return\n")
```

```
+   print(aveReturn)
+ }
> gr
Time Scale: month

Average Return
       AAPL        MSFT         RHT
0.030091603 0.001532898 0.020646707
```

It would be useful to also create `summary` and `plot` methods for the `"stockReturns"` class. To do so, functions `summary.stockReturns` and `plot.stockReturns` would be written.

This brief introduction to S3 classes is likely sufficient for many beginning and intermediate users. The topic of classes and methods will come up one more time in this tutorial: in the name space of a package.

Readers interested in additional resources and instruction for S3 classes may review Leisch (2009), Development Core Team (2012b), and Chambers (2008). Advanced programming projects may require S4 classes, for which many helpful resources are available: Genolini (2008), Bates (2003), Development Core Team (2012b), and Chambers (2008, p364-368).

4 Package building mechanics

The mechanics of creating an R package are the easiest step in making a useful package. In fact, it is useful to bundle code for any research project into an R package to support basic documentation of data and functions regardless of whether that package is to be shared with others. As more data sets and functions are identified or created, they may be added to the package without substantial effort.

Before building a package, upgrade to the latest version of R. Doing so will help ensure that the latest quality tools and resources are available. A working version of LaTeX (e.g. MacTeX[6] or MiKTeX[7]) is also required for users of any platform who wish to formally check a package or apply additional UNIX commands described in Section 4.3.1. This check step is recommended for internally used packages and required for a package being submitted to CRAN.

There are a couple additional steps to preparing a Windows machine for building R packages. The most basic step is to install *Rtools*.[8] During this installation, some users may also be required to specify the System path variable, which is described in the footnote.[9]

[6] http://www.tug.org/mactex/

[7] http://miktex.org/

[8] http://cran.r-project.org/bin/windows/Rtools/

[9] The following paths, or their equivalents, should be included in the System variables: (this footnote continues on next page)

```
C:\Program Files\R\R-2.14.2\bin\x64;C:\Rtools\bin;C:\Rtools\gcc-4.6.3\bin;
```

There should be no spaces following any semi-colons in the list of paths. On Windows 7, the System variables may be set by going to *Control Panel* > *System and Security* > *System*. Next, go to the left-menu option *Advanced system settings*, the *Advanced* tab in the pop-up window, and then the *Environment Variables* button. Another pop-up window provides the System variables at the bottom. Scroll to the *Path* variable, select it, and click *Edit*. If necessary, add the paths provided above. If necessary, adjust `2.14.2` and `x64` in the first path provided, and adjust

Sections 4.1-4.3 describe the three steps necessary to construct a package from existing code and data:

1. Generate the package template using the `package.skeleton` function.
2. Fill in details of the package in the template. Often times this step is repeated as files require revision based on warnings or errors in step 3.
3. Install the package, build a tarball to facilitate sharing the package with others, or formally check the package for submission to CRAN or another repository.

Section 4.4 outlines how to expand an existing package with new functions or data sets, and Section 4.5 describes how to merge help files and provides a summary of additional files that may be contained in packages.

4.1 Creating a package template

Creating a package template in R is generally quite easy. Start with a clear R session (e.g. run `rm(list=ls())` to clear the global environment), load in each function and data object to be included in the package, run the `package.skeleton` command, and then find where the package files were generated:

```
> load("path/to/fileWithAllFunctionsAndObjects.RData")
> package.skeleton("packageName")
Creating directories ...
Creating DESCRIPTION ...
Creating Read-and-delete-me ...
Saving functions and data ...
Making help files ...
Done.
Further steps are described in './packageName/Read-and-delete-me'
> getwd()
[1] "/Users/ddiez"
```

The first line is one way to get all objects into R, though there are numerous other ways (e.g. copy and paste functions into the console) that produce equally valid results. The `package.skeleton` function creates a new folder with 2-3 files and 2-3 folders, each with additional files. If the current R session contains more objects and functions than should be included in the package, use the optional argument `list` in `package.skeleton` to specify those objects to include in the package. Additionally, use the `path` argument to place the package files in a location other than the current working directory. Above, no path was specified, so instead `getwd` was used to learn in which directory the package files were generated.

---

gcc-4.6.3 as necessary. MiKTeX will usually update the *Path* variable to ensure its tools are accessible. If its tools are not accessible (when running `R CMD check`, as described in Section 4.3.1), add the following path to the *Path* variable:

`C:\Program Files (x86)\MiKTeX 2.9\miktex\bin\`

Below are the files and folders commonly created by `package.skeleton`:

DESCRIPTION (file). This file summarizes information about the package. It should be viewed and updated in a simple text editor (e.g. TextEdit or Notepad).

NAMESPACE (file). This file, now automatically generated by R versions 2.13.0 and later (Development Core Team, 2012a), manages package interactions. This file should be viewed and updated in a simple text editor. It is now required for all CRAN submissions.

Read-and-delete-me (file). This second file contains basic package building directions and may be deleted or stored elsewhere.

man (folder). A folder containing manual (help) files that have a file extension *.Rd*. These files take a form not unlike LaTeX. Importantly, generated help files are customized to the structure of each function and data object.

R (folder). A folder containing a *.R* file for each function in the package. Functions in a package are edited by modifying the functions listed in these files. Any updates to a function name or arguments should also correspond to updates in the help file, notably the `\usage` section.

data (folder). A folder with the data objects in the package. Each data object is saved in this folder under its own *.rda* file.

The *DESCRIPTION*, *NAMESPACE*, manual, and function files are edited directly. If no functions or data objects are included, then the *R* or *data* folder will not be created. However, these folders may be manually created later if functions or data objects are added to the package (see Section 4.4).

4.2 Editing package documentation

R generates skeleton files that require attention. These files should be edited in a simple text editor, such as TextEdit, Notepad, or R's code editor.

4.2.1 DESCRIPTION file

The initial *DESCRIPTION* file, shown below, contains five sections that require special attention.

```
Package: PackageName
Type: Package
Title: What the package does (short line)
Version: 1.0
Date: 2012-03-21
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?
```

The *Title*, *Version*, *Author*, *Maintainer*, and *Description* sections are self-explanatory. The *License* entry for most packages on CRAN is *GPL-2* or *GPL-3*, which refer to the General Public License 2 and 3, respectively. Updates from version 2 to 3 are discussed in Free Software Foundation (2010). Other options (e.g. *GPL (¿= 2)*, *Unlimited*) also exist, but are less commonly used. It is the responsibility of authors to review these licenses and make an informed decision about which is most appropriate. The remaining entries in the *DESCRIPTION* file may also be edited, as necessary.

Package dependencies require a new field in the *DESCRIPTION* file called *Depends*:

```
Depends: grDevices, graphics, stats, utils, R (>= 2.10)
```

Nearly all packages rely on at least one of the packages above, which are usually loaded automatically on R's startup. For packages that require a specific version of R, also specify the minimum version as an additional entry in the *Depends* field.

See Chambers (2008, p94) for additional fields and options.

4.2.2 NAMESPACE file

Packages that are to be submitted to CRAN must include a *NAMESPACE* file (no file extension). This file helps maintain stability among interactions with other packages. The name space file is generated by `package.skeleton` in R versions 2.13.0 and later (Development Core Team, 2012a). If the package was generated without a name space, then create a text file called *NAMESPACE* in the package's main directory, and verify after the creation of the file that no extension has been added by the text editor.

The name space file specifies which functions in the new package are to be accessible to end-users and what packages must be imported. The R-generated name space file will automatically make all functions available; a manually-generated name space can also make all functions available by adding the following line to *NAMESPACE*:

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

Alternatively, this line may be deleted (or not added), and functions that are to be available to users can be specified using `export` and listing all functions to be made available in parentheses:

```
export(funcName1, funcName2, funcName3)
```

Any packages that also have a name space (now nearly all packages) and are required by the new package should be listed in an `import` command:

```
import(stats, utils, graphics, grDevices)
```

Packages without a name space should not be listed here.

Lastly, new S3 methods must be specified using the `S3method`. For example:

```
S3method(print, newClassName)
```

There are many additional options within the name space, but they are not discussed here. For additional reading, see Chambers (2008, p103) and Development Core Team (2012b).

4.2.3 Help files

Help files are structured in a style that will be familiar to LaTeX users. For users unfamiliar with LaTeX, commands are started with a backslash, and commands are typically applied to text within braces. A percent sign is used to comment out the remainder of a line.

There are 14 components in the template for function help files and 12 in data help files: 2-3 brief, stand-alone commands at the top of each template, followed by several multiline commands that denote sections (e.g. `\title`, `\description`). The `\name`, `\alias`, and for data help files, `\docType` commands at the top of the help files should be left as-is unless there are good reasons to make modifications (see Section 4.5.1).

It is advisable to fill in as many of the sections in the help file as possible. Theoretically, only the `\name`, `\alias`, `\title`, and `\description` sections are required and all others may be deleted, however, providing only this very limited information would make "help file" a misnomer.

Developers should pay special attention to the `\examples` section, which can be especially useful for making a data set, function, or method accessible to the end-user. There is a temptation to misinterpret this section as equivalent to the verbatim environment in LaTeX. However, all percent symbols must be escaped (i.e. to generate `%`, write `\%`). Additionally, some examples may require certain user interaction or an internet connection; including these commands as-is in the examples will generate warnings or errors when checking the package. Rather, any code that should not be run during a formal package check should be enclosed in a `\dontrun` command. Additionally, if there are commands to be run but not shown, the `\dontshow` command should be employed.

Several extra commands in help files are available to create modified text, linking, and adding equations and figures.

- Emphasize (italicize) or bold text using the `\emph{emphasize}` and `\bold{bold}`.
- Monospaced text can be generated via `\code{monospaced text}`.
- Help files within a package may be linked using the link command: `\link{otherFcn}`; this example prints "otherFcn" and links to the help file that contains `\alias{otherFcn}`. To link to a function or data object in another package, for example the `map` function in the `maps` package, use `\link[maps]{map}`. Wrap the `\link` command inside a `\code` command to show a link in monospaced text: `\code{\link{otherFcn}}`. Note that reversing the order of the `\code` and `\link` commands will produce an error.
- Link to a URL using `\url{http://www.jstatsoft.org/}` or by creating linked text as `\href{http://www.jstatsoft.org/}{Journal of Statistical Software}`. Emails may be contained in `\email{david@openintro.org}`.
- Developers may implement mathematics or equations in documentation using LaTeX notation with the following two commands: `\eqn{inline math here}` and `\deqn{math on own line}`. These commands will create some symbols and mathematics in the standard HTML help files, while full support for mathematics is reserved for the package's PDF manual. Note that neither command supports multi-line equations.
- Figures may be included using the `\figure{figurename}{alternative text}` command. The figure name should include the extension and the figure should be placed in a folder called *figures* in the *man* folder. (Figure support in help files is supported in R version 2.14.0 and later (Development Core Team, 2012a).)

For additional information on help files and a complete list of available help file commands, see Development Core Team (2012b, Chapter 2).

4.3 Shell commands for R packages

After a package's files have been initialized and edited, the package may be installed, built, and checked using shell commands. Installing a package in this way is not meaningfully different than installing from source using the `install.packages` command in the R console. Building a package means to put the package source files in a *tar.gz* file (a.k.a. a tarball), which is a useful way to share package source files. Checking a package means to perform automated checks that evaluate whether there are obvious errors in the package, e.g. identification of invalid commands in documentation and verifying examples run.

I recommend copying the package folder to the *Desktop* to simplify file navigation. Shell commands may be performed in the *Command Prompt* application on Windows (*Start* > *All Programs* > *Accessories* > *Command Prompt*) or the *Terminal* application on Mac OS X (*Finder* > *Applications* > *Utilities* > *Terminal*).

4.3.1 R CMD install, build, and check

The first step is to open a shell prompt and ensure it sees the package files. To learn the present working directory, type `pwd` and hit *return*. From here, a user may navigate to the directory containing the package using the directions in the footnote.[10] Alternatively, it may be easier to create a copy of the package in the directory returned by `pwd`.

When a shell prompt is pointed at the directory with an R package, there are several available options. Here three of those options are considered: install, build, or check the package. Each of these is accomplished via `R CMD command PackageName`, where `command` can be `install`, `build`, or `check`. For example, below a package called `PackageName` has been installed:

```
David-Diezs-MacBook-Pro:~ ddiez$ pwd
/Users/ddiez
David-Diezs-MacBook-Pro:~ ddiez$ R CMD install PackageName
* installing to library '/Library/Frameworks/R.framework/Versions/2.14/
Resources/library'
* installing *source* package 'PackageName' ...
** R
** data
** preparing package for lazy loading
** help
*** installing help indices
** building package indices ...
** testing if installed package can be loaded

* DONE (PackageName)
David-Diezs-MacBook-Pro:~ ddiez$
```

Had I chosen to perform a `R CMD build PackageName` command, then a tarball with the package source would have been generated in the present working directory. The `R CMD check PackageName` command would have performed many formal checks on the package; this command usually takes significantly longer than the other two commands to run, and it often returns warnings and errors with concise explanations about what went wrong. Packages submitted to repositories should always undergo a formal check, and all warnings and errors must be addressed before submission. That is, if a submitted package returns a warning or error from `R CMD check`, then it is generally returned to the user for revision. Submissions to CRAN undergo a slight variation of `R CMD check`, and users may check their packages in the same way CRAN does by using the following command, new as of R version 2.14.2 (Development Core Team, 2012a):

```
R CMD check --as-cran PackageName
```

Note that if a package is being installed or checked from a tarball, the full file name must be provided. For example, if the tarball file name is *PackageName_1.0.tar.gz*, then the following command would be required to install this package:

```
R CMD install PackageName_1.0.tar.gz
```

[10] For this purpose, two additional commands would be useful: `ls` and `cd`. The `ls` command prints out the folders and files that are in the present working directory. The `cd` command allows a user to navigate around, e.g. type `cd ../` to move up one directory or `cd foldName` to go into a folder named *foldName*.

4.3.2 Building a binary and PDF manual

Additional shell commands may be used to generate binaries for a particular platform or create a PDF manual for a package. The commands to perform each of these actions are listed below in their respective order:

```
R CMD INSTALL --build packageName
R CMD Rd2pdf packageName
```

The command used to generate binaries was changed in R version 2.14.0 (Development Core Team, 2012a).

4.4 Adding functions and data to an existing package

Adding a function to a package requires the generation of two files. First, write the function in a *.R* file where the file name is the same name as the function, e.g. a function called `myFunction` would go in a file called *myFunction.R*. Next, load the function into an R session using `load` or by copying and pasting the function, and apply the `prompt` command to `myNewFunction` to generate a help file template:

```
> prompt(myNewFunction)
Created file named 'myNewFunction.Rd'.
Edit the file and move it to the appropriate directory.
```

To learn the current working directory, apply the `getwd()` command. The generated *.Rd* file will be in this location. As directed, move the new help file to the *man* folder in the package and update its contents.

The steps to add a new data object to a package are equally simple. First, load the object into an R session, save the data object to an *.rda* file using the `save` function, then apply the `prompt` command to make a new help file template:

```
> myDataObject <- read.delim("myData.txt", TRUE)
> save(myDataObject, file="myDataObject.rda")
> prompt(myDataObject)
Created file named 'myDataObject.Rd'.
Edit the file and move it to the appropriate directory.
```

Finally, move the data and help files to the *data* and *man* folders, respectively, and update the help file.

Functions and data objects may be removed from a package by removing their files from the *data*, *man*, and *R* folders. Appropriate updates should also be made to the *NAMESPACE* file.

4.5 Additional package building topics

4.5.1 Merging help files

It is common to merge help files or related functions and, occasionally, data files. For an example of an elegantly merged help file for multiple functions, see the help files for `sub`, `glm`, and `head`. While some authors may choose to provide documentation for methods, it may also be reasonable to merge these help files with the function generating those objects.

Help files may be merged in just a few steps. First, choose a help file that will serve as the "master" help file. Second, create an `\alias` entry for each item that should reference that help file, e.g. copy and paste the `\alias` command from each of the other help files into the master help file. Third, if the master help file is for multiple functions, include a `\usage` command for each and ensure all arguments are documented. Finally, update the title, description, and other text to supply documentation for all objects.

If the help files for methods of a function's output are being merged with that function, and there are no nuances about the methods that should be communicated to the end user, it is reasonable to simply add the `\alias` command for each method.

4.5.2 Optional package materials

Packages may house many more files and folders than are discussed in this guide, though these files and folders are not required for the majority of packages, though some packages may benefit from their use. Below is a brief list of these items.

INDEX (file). Briefly describes each function and data object of importance.

configure (file). See Development Core Team (2012b, Section 1.2).

cleanup (file). See Development Core Team (2012b, Section 1.2).

LICENSE or LICENCE (file). Contains a copy of the license for the package *only if it is a non-standard license*.

NEWS (file). A file indicating updates for a package that follows GNU coding standards.

demo (folder). A folder for scripts that demonstrate some of the capabilities of the package. Demos are run by the end-user via the `demo` function.

exec (folder). Additional executable scripts should be placed in this folder.

inst (folder). Create a special citation or include extra files or folders to be copied to each user's installation directory. A special citation for a package may be generated using a *CITATION* text file (no extension) in the *inst* folder; details are available in Development Core Team (2012b, Section 1.10). File and folder names in the *inst* folder cannot overlap with standard file or folder names in packages listed here or on page 13 of this guide.

po (folder). A folder containing files relevant to internationalization. See Development Core Team (2012b, Section 1.9).

src (folder). Source code for `C`, `C++`, `FORTRAN`, or other non-R code is placed in the *src* folder. Note that calls to functions such as `.C` in a package must also include a `PACKAGE` argument. Additional details for including source code in packages are provided in Development Core Team (2012b, Chapters 1 and 5).

tests (folder). Test code for the package in the form of *.R* files are included in this directory.

For a complete review of implementation using these files or folders, see Development Core Team (2012b, Chapter 1).

5 Discussion

Packages that satisfy `R CMD check` without warnings or errors may be submitted to CRAN or another repository. For submission to CRAN, follow the directions provided on the CRAN homepage (R Development Core Team, 2012):

> To "submit" to CRAN, check that your submission meets the CRAN Repository Policy, 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.

The upload should be a tarball (see Section 4.3.1) that has been checked via `R CMD check`. To upload a file, a file transfer protocol (FTP) application is required; any functioning FTP application should be sufficient. Two free options are listed in the footnote[11] for users currently without such an application.

Two supplements to this tutorial have been provided online: screen-capture videos and lecture slides. The videos are available on RFunction.com via YouTube and follow the package building mechanics discussed in Section 4. Lecture slides (PDF and Beamer source) are also provided at RFunction.com and are built for a 60-90 minute R packages talk. Much lecture material is optional and could be eliminated for a shorter talk.

Numerous other resources also exist online, many of which have gone to great use in the construction of this guide. *Writing R Extensions* is *the* R package building guide with substantial detail and discussion and is freely available on CRAN (Development Core Team, 2012b). *Creating R Packages: A Tutorial* is an alternative, freely available tutorial that provides an excellent introduction to classes, methods, and the basics of package building; however, some material has become outdated due to the evolving nature of R (Leisch, 2009). *Software for Data Analysis* is a book resource providing excellent guidance for general programming in R, an extensive discussion of classes and methods, package building guidance, and many other helpful topics (Chambers, 2008).

[11] Mac OS X: Cyberduck (cyberduck.ch). Windows: WinSCP (winscp.net/eng/index.php).

Acknowledgments

This guide and its supplements would not have been possible without the helpful feedback and input of Travis Gerke and Christopher Barr, the many questions that my colleagues have asked that I could not initially answer, and the many excellent resources that so many R users before me have made available online.

References

Bates D (2003). Converting packages to s4. *R News*, 3:6–8.

Chambers JM (2008). *Software for Data Analysis: Programming with R*. Springer Verlag, New York.

Dalgaard P (2008). *Introductory Statistics with R*. Springer Verlag, New York.

Development Core Team (2012a). R news.

Development Core Team (2012b). *Writing R Extensions, version 2.14.2*.

Diez D and Christou N (2012). *stockPortfolio: Build stock models and analyze stock portfolios.* R package version 1.2.

Diez DM (2012). *Building R Packages (lecture slides)*.

Fellows I (2011). *Deducer: Deducer*. R package version 0.5-1.

Fox J (2005). The R Commander: A basic statistics graphical user interface to R. *Journal of Statistical Software*, 14(9):1–42.

Free Software Foundation (2010). A quick guide to GPLv3 - GNU project - free software foundation (FSF).

Genolini C (2008). A (not so) short introduction to s4. *Comprehensive R Archive Network*.

Leisch F (2009). Creating r packages: A tutorial. *Comprehensive R Archive Network*.

Peng RD (2002). *An Introduction to the Interactive Debugging Tools in R*.

R Development Core Team (2012). The comprehensive r archive network.

Venables WN, Smith DM, and R Development Core Team (2012). *An Introduction to R*. ISBN 3-900051-12-7.