Package as Analysis

Joshua H. Cook

2019-03-17

Contents

Welcome

This is a manual on how to use the standard R package framework for data analysis. Though potentially more work, especially at the start, the purpose of using the R package framework is to maintain a clear and reproducible analysis.

[This book is currently in progress, though any feedback is welcome.]

Why

Why bother with maintaining a package framework while also doing data analysis? It is a great question, especially when one considers the complexity and fluidity of an analysis and the rigidity of the R package framework. The answer is that some rigidity is needed - but just the basics. That is what the R package framework provides. There is a place for everything, though sometimes getting it to work (ie. build and pass checks) requires a few extra steps.

During my own analyses, I found things were getting much too disorganized and decentralized. What would start out as exploratory would morph into a subdirectory graph more complex than the oringinal parent analysis. Perhaps for more organized people, this is unnecessary, but for those of us who want order and aren't sure how to get it, this framework offers a great place to start.

The final (more abstract) reason for using the R package framework is to battle the current issue of reproducibility. Reproducibility is the cornerstone of science - if a finding it true, anyone should be able to replicate it. However, the scientific community has been dealing with an astounding amount of irreproducibility, most famously documented by the Open Science Collaboration. If the analysis is organized as an R package, though, an analysis can be re-run entirely by anyone else familliar with R. Thus, whether they are collaborators and competitors, anyone should be able to follow the analysis a scientist publishes.

Advantages

There are many advantages to using this framework. Here are just a few, though I am sure you will find there are many others:

- Because this is a standard framework, other will be able to navigate the directories and files adeptly.
- The implementation of tests on functions and subroutines will make bugs easier to find and increase overall confidence in the validity of the analysis
- This is a seamless mixture of scripts and markdown files for the separation of functions and analysis
- Complete documentation of functions makes returning to code later much easier!
- The analysis can take advantage of normal R package tools such as Travis-CI and Codecov integration, pkgdown, and devtools (build checks, documentation, etc.).

Examples

[coming soon] Allele-specific *KRAS* copy number alteration

About

About this Book

[TODO]

About the Author

I am a classically-trained biologist-turned computational biologist. I graduated with degrees in Molecular Biology and Biochemistry, and Chemistry from the University of California, Irvine in 2017. My research focused on investigating the patterns and mechanisms of dissemination by which *Toxoplasma gondii*, an obligate, intracellular parasite, infects a human host Cook *et al.*, 2018. I started my graduate studies at Harvard Medical School in 2018, and after rotating in a chemical biology lab and a *Vibrio cholerae* lab, I finally decided to study cancer using computational biology. Since then, and continuing still today, I have been learning computer programming and statistics, trying to catch up to my peers. Consequently, I have fallen in love with R, especially because of the Tyidyverse and tidy data.

Resources

The best resource for making R packages is R Packages by Hadley Wickham.

There are some useful R packages you will want to have installed:

- `devtools` - will do most of the development building and checking
- `roxygen2` - makes all of the documentation
- `usethis` - for preparing all of the pieces and tools you want to include
- `testthat` - for running tests
- `kintr` - for compiling all of the Rmarkdown files

These can be installed using the following code.

```
install.packages(c("devtools", "roxygen2", "usethis", "testthat", "knitr"))
```

License

This work is under a Attribution-NonCommercial-ShareAlike 4.0 International License (CC BY-NC-SA 4.0)

Chapter 1

Framework

1.1 Introduction

I have decided to introduce the R packagge framework before going through the set-up process because, quite frankly, I think this is more important. There are many sites that outline the steps of creating an R package (they likely do it better than I will next chapter), but the main point of this book is how to make the R package into a data analysis project. Therfore, understanding the use of the framework should be the reader's focus, here.

I try to adhere to the following process for each peace of the framework:

1. what is it and what is its main role?
2. when does it need to be used or adjusted?
3. what is unique about its use for a data analysis project?

1.2 Vignettes

I am starting with vignettes because they are the heart of the data analysis. Ironically, they are often covered last in tutorials on creating R packages, and rightly so! For an R package, the vignettes merely show the functionality of the package. They are usually user guides or are attempts to inspire potential users.

However, in a data analysis, the vignettes *are* the analysis. They are where ideas are formed, discussed, tested, and validated. Everything is centered around a vignette. If you already use R markdown for data analysis, then this should be of little perturbance to your standard operating procedure. If you do not, though, this should be a welcomed change.

1.2.1 Starting a Vignette

Ideally, each vignette will be a different step in the analysis. Alternatively, each vignette could be a different part of the project (e.g. analyzing the results of a screen in one vignette and analyzing the results of validation experiments in another). However you choose to divide up the analysis is your choice (though I have found R markdown files to slow down RStudio if they get too long or image intensive). To begin a vignette, use the usethis function, passing the name of the vignette.

```
usethis::use_vignette("A01_part1_firstvignette")
#>  Setting active project to '/path/to/pkg/ExampleAnalysisPackage'
#>  Adding 'knitr' to Suggests field in DESCRIPTION
#> Setting VignetteBuilder field in DESCRIPTION to 'knitr'
#> Adding 'rmarkdown' to Suggests field in DESCRIPTION
#> Creating 'vignettes/'
#> Adding '*.html', '*.R' to 'vignettes/.gitignore'
#> Adding 'inst/doc' to '.gitignore'
#> Creating 'vignettes/A01_part1_firstvignette.Rmd'
#> Modify 'vignettes/A01_part1_firstvignette.Rmd'
```

Notice that the ".Rmd" extension is ommited in the above function - usethis adds the appropriate extensions in most cases.

It is important to deliberately decide on a naming scheme. I have opted for "(A-Z)(##)_part_subpart" where the leading alphanumeric index is made of two levels followed by the names of the two levels. The first letter is the highest level of orgnization and the second two digits are for the sublevel. For the naming, the first name ("part") is the name referring to the layer of organization dictated by the first letter of the leading alphanumeric index. The second name ("subpart") is referring to the layer of organization dictated by the second two digits of the leading alphanumeric index

Therefore, the follow-up analysis that follows "A01_part1_firstvignette" could be "A02_part1_secondvignette". A vignette for another part of the project would alternatively be called "B01_part2_firstvignette". I now have two branches of analysis in my project, the first with two pages of analysis and the second with only one.

Of course, you should customize the naming scheme if this does not work for you, but it is ddefinitely important to have one and to keep it ordered. The only limit is that the vignette name cannot start with a number.

1.2.2 Formatting a Vignette

[TODO]

1.3 The R/ Directory

1.3.1 Normal Use

The R directory will hold all R scripts (usually files ending with the ".R" extension). For a normal package, this is all of the source code that builds the package.

1.3.2 For Data Analysis

For the purpose of a data analysis project, this will still hold R scripts (with the ".R" extension), but the organization of scripts is different.

Chapter 2

Data Management

This is the most annoying part of a large data analysis project. Usually, the goal is to stike a balance between complexity and ease-of-use. The R package framework does most of the work, but there are a few finicky details.

The data structure is divided between 3 directories at the project root:

1. **data-raw** - messey and raw data that needs a bit of work to be ready for use in the vignettes
2. **inst/extdata** - data that is analyzed in the vignettes; is usually not tidy but not too messy either
3. **data** - *tidy* data that is meant to be reusable and is important to the project

The order listed above is from least to most prepared. Each has a role and purpose for existing.

2.1 The "data-raw" folder

2.1.1 When to use it

The "data-raw" directory is meant to hold the messy and unorganized data that needs a bit of help to get to a usable state. Therefore, the data stored there should be accompanied by a R script that puts them into the simplest usable state. I think it is best practice to have an individual script for each data type because it is meant to be simple and easy to understand. The main idea is to do the fewest steps possible to get the data into a slightly more useful state and stored in "inst/extdata".

Most data can likely skip "data-raw" and go directly into "inst/extdata". As you can tell from the previous paragraph, the decision between putting data into "data-raw" and "inst/extdata" comes down to one's definitions of "messy" and "usable". However, there is one case (at least that I have come across) where using data-raw is required (i.e. `R CMD check` throws an error is you do not).

The one case where the data must be stored in "data-raw" and then preprocessed to "inst/extdata" is if it is a lot of small data files that will be joined together. If the package is built (using `devtools::build()`), a warning for each file (which can potentially be a lot of files!) will be thrown saying it is bad to compress all of these small files.

```r
devtools::install()
#> ...
#> Warning in utils::tar(filepath, pkgname, compression = "gzip", compression_level = 9L,  :
#>      storing paths of more than 100 bytes is not portable:
#>      "ExampleAnalysisPackage/path/to/offending/file.txt"
#> ...
```

To avoid this, I read all of the data into a single object (usually a list) and then save that to “inst/extdata”.

2.1.2 Example

Since the R script is in “data-raw”, it will not be built with the rest of the package. Therefore, it is important to clearly state the intedned working durectory. To stay consistent, I recommend staying in the package’s root directory.

However, the common case where I will use “data-raw” is if I have a folder with a lot of small data files that will be joined together.

Chapter 3

Getting Started

Setting up the package is mostly automated and is well documented in R Packages by Hadley Wickham. If you are running the analysis on your local machine, I would recommend using RStudio (which you likely already do), but this is possible to do on a remote computing cluster (which is how I work). I begin by going through the steps of setting up the basic package framework, which is the same for local or remote work. Following that, there is a section for how I work remotely. This can be skipped if you only work locally or if you already have a system you enjoy (though I highly recommend the system I currently use). I finish off with a few extras that I recommend using, but are not necessary.

3.1 Setting Up a R Package

The set up process is rather simple. If using RStudio, you can start a new R project as a package. Otherwise, the following command will get the basic framework started. There is a lot of overlap between the devtools and usethis package. I believe that RStudio is trying to fade out devtools and instead have people use the various packages that were split from it, including usethis.

The advantage to using the usethis functions for seemingly simple tasks (such as making the "data-raw" directory) is that it will also add the necessary lines to ".RBuildignore," the "DESCRIPTION," and "NAMESPACE" if needed.

3.1.1 The Basics

It's easy to create the package from the R console, just replace the directory path in the following example with your desired directory. Usethis will *creat* the "ExampleAnalysisPackage" directory - do not make this directory beforehand.

```r
usethis::create_package("/path/to/pkg/ExampleAnalysisPackage")
#>  Setting active project to 'p/path/to/pkg/ExampleAnalysisPackage'
#>  Creating 'R/'
#>  Creating 'man/'
#>  Writing 'DESCRIPTION'
#>  Writing 'NAMESPACE'
#>  Changing working directory to '/path/to/pkg/ExampleAnalysisPackage'
```

You can then add a license as shown below. I generally use a GPL-3, though you can get a lot of information on the common licenses at choosealicense.com.

```
usethis::use_gpl3_license(name = "Your Name")
#>   Setting active project to '/path/to/pkg/ExampleAnalysisPackage'
#>   Setting License field in DESCRIPTION to 'GPL-3'
#>   Writing 'LICENSE.md'
#>   Adding '^LICENSE\\.md$' to '.Rbuildignore'
```

Prepare the project to use roxygen for documentation.

```
usethis::use_roxygen_md()
#>   Setting Roxygen field in DESCRIPTION to 'list(markdown = TRUE)'
#>   Setting RoxygenNote field in DESCRIPTION to '6.1.1'
#>   Run `devtools::document()`
```

Create a README file. You can also opt to use a normal Markdown file with `usethis::use_readme_md()`, though I would recommend to just go with an R Markdown file.

```
usethis::use_readme_rmd()
#>   Writing 'README.Rmd'
#>   Adding '^README\\.Rmd$' to '.Rbuildignore'
#>   Modify 'README.Rmd'
```

Create a "NEWS" file for announcing major changes to the project.

```
usethis::use_news_md()
#>   Writing 'NEWS.md'
#>   Modify 'NEWS.md'
```

Create a "data-raw" directory.

```
usethis::use_data_raw()
#>   Creating 'data-raw/'
#>   Adding '^data-raw$' to '.Rbuildignore'
#> Next:
#>   Add data creation scripts in 'data-raw/'
#>   Use `usethis::use_data()` to add data to package
```

Finally, set up the use of testthat package for testing.

```
usethis::use_testthat()
#>   Adding 'testthat' to Suggests field in DESCRIPTION
#>   Creating 'tests/testthat/'
#>   Writing 'tests/testthat.R'
```

Note: If you are working remotely (i.e.. SSHing to the computer running the code), many of the usethis functions will open the file that you just asked them to create (e.g.. open "NEWS.md" after using `use_news_md()`) in vim. To suppress this, just pass the parameter `open = FALSE`. Otherwise, it is set to `interactive()`.

3.1.2 Git and GitHub

If you are programming, you should be using git. This is especially important in the sciences because git logs can be used to resolve legal conflicts and issues of data falsification. GitHub is not essential, though I would highly recommend you use it because it makes managing files and collaboration much easier. It is also essential for taking advantage of some of the best parts of an R package such as build checks and `pkgdown` (see Extras below)

To get started with git, there are *tons* of resources available, so I will not describe it here. If you are new to git and GitHub, here are a few good resources to get you started:

- An introduction to Git: what it is, and how to use it
- How To Use Git: A Reference Guide
- GitHub Guides: Hello World

There is a usethis function to initiate git (`usethis::use_git()`) though I always prefer to set up myself.

```
usethis::use_git()
#>   Initialising Git repo
#>   Adding '.Rhistory', '.RData', '.Rproj.user' to '.gitignore'
#> OK to make an initial commit of 8 files?
#> 1: Yeah
#> 2: Absolutely not
#> 3: No way
1
#> Selection: 1
#>   Adding files and committing
```

3.2 Working Remotely

If you conduct work remotely, I'm going to assume that you have SSH set up and running. Otherwise, there are plenty of resources available, and you should review the material available by your system admin.

Though I prefer RStudio for normal package development, I spare my computer the pain of performing complex and heavy computation, opting instead to off-load it to the Harvard Medical School Research Computing Cluster. Therefore, I use SublimeText3 as by text editor and send code to the remote computing node over SSH using iTerm2 as my terminal. Finally, I use SSH File System (SSHFS) to "mount" the remote directory to my local directory.

3.2.1 SublimeText3 Set-Up

Here are the handful of SublimeText3 (ST3) packages I use for R coding, followed by any particular notes on their use:

- LSP - "Gives Sublime Text 3 rich editing features for languages with Language Server Protocol support"
- MarkdownEditing - "Markdown plugin for Sublime Text. Provides… more robust syntax highlighting and useful Markdown editing features for Sublime Text."
- R-IDE - "[A]iming to utilize the use of language server + better support R Markdown + better support of R packaging + …"
- SendCode - "Send code and text to macOS and Linux Terminals, iTerm, ConEmu, Cmder, Tmux, Terminus; R (RStudio), Julia, IPython."

LSP and R-IDE handle syntax and completion in ST3. It isn't a great system, so if you know of a better set-up in ST3, please let me know. MarkdownEditing and R-IDE combine to make R Markdown feasible. The SendCode package essentially copies, pastes, and runs my code written in ST3 to the terminal when I press `command` + `return`. This way, I can type in ST3 and run in the terminal without using the mouse.

Before moving on, I made this snippet to quickly add a code chunk.

~~~
<snippet>
        <content><![CDATA[
```{r ${1:chunk_name}}
$0
```
]]></content>
        <!-- Optional: Set a tabTrigger to define how to trigger the snippet -->
~~~

```
        <tabTrigger>rchunk</tabTrigger>
        <!-- Optional: Set a scope to limit where the snippet will trigger -->
        <scope>text.html.markdown.multimarkdown, text.html.markdown</scope>
        <description>create a Rmd code chunk</description>
</snippet>
```

3.2.2 Using SSH File System

I use SSH File System (SSFHS) to "mount" my remote directory to my local directory. It is essentially SFTP and SSH combined (the details go right over my head) and it is fairly easy to get set-up. Here is a link to get everything going, and I have included the steps I used below.

To start I downloaded and installed FUSE for macOS. Then I downloaded and installed the latest build of SSHFS. Finally, I created an empty directory that will become the place that I mount the remote directory. Typically, I make the root of the package, though I could see instances where you would want the package in a subdirectory below it.

```
# on local
mkdir ~/local/path/pkgName
```

On the remote server, I make a directory with the same name

```
# on remote
mkdir /remote/path/pkgName
```

Finally, to connect the two, I use the following command that is pretty much identical to initiating a normal SSH session.

```
# on local
sshfs username@remote.host.com:/remote/path/pkgName ~/local/path/pkgName
```

Now, the computer will treat the mount just like a normal flash drive, and ST3 fully accepts it. The only change I made to ST3 was to map a key-binding to "Project/Refresh Folders". This way, if new files are created remotely, a quick key-stroke and everything is visible in the ST3 sidebar.

3.2.3 Git and GitHub

If working remotely, I have found it much easier to handle everything git-related on the remote side. Therefore, I created SSH RSA keys and shared the public one with the GitHub repository so I could push over SSH. Setting this up is pretty simple and well outlined in Connecting to GitHub with SSH.

3.3 Extras

Though these next few items are not required, I *highly recommend* implementing them because they each take advantage of the fact that this project adheres to the standard R framework. Their different functions are all reasons to go through the trouble of maintaining this framework.

3.3.1 pkgdown

Pkgdown ties a bow around your package, slaps it on the bottom, and builds a gorgeous and professional website rich with useful features. It builds the documentation for easy reference, presents the vignettes, and organizes all of the package meta-data so it is easily viewable and understandable. Here are some packages that take advantage of pkgdown:

- pkgdown (of course)
- ggplot2
- ggsci
- ggasym (a shameless plug of my own lil' package)

The use of pkgdown obviously begins with a usethis function.

```
usethis::use_pkgdown()
#>  Modify '_pkgdown.yml'
#>  Adding '^_pkgdown\\.yml$' to '.Rbuildignore'
#>  Creating 'docs/'
#>  Adding '^docs$' to '.Rbuildignore'
```

All that you have to do from there is use `pkgdown::build_site()` to build the site whenever the project is at a good stopping point for the day. (If working remotely, pass `preview = FALSE` to prevent pkgdown from searching for a browser to display in when completed.)

To show the website on GitHub, go to “Settings” in the repository, and select “master branch /docs folder” from the options in the “GitHub Pages” section. It should look something like this (another shameless plug for lil' ole' ggasym).

3.3.2 Travis-CI, Appveyor, and Codecov

Note: To use these, you must have a GitHub repository that you have push to.

GitHub integration also opens up the use of continuous integration (CI) apps. Travis-CI and Appveyor are useful for checking the build status of the package. I just use both because they each require so little effort to integrate and each provides their own suite of functions. Notably, Appveyor build the package on Linux and Windows. To get started, just use usethis.

Codecov provides an indication as to how well the package's tests cover the code. Though not a perfect measure of test quality (nothing ever will be), I find this tool to be helpful for me to find which functions I have and have not created tests for.

```
usethis::use_travis()
#>  Setting active project to '/path/to/pkg/ExampleAnalysisPackage'
#>  Writing '.travis.yml'
#>  Adding '^\\.travis\\.yml$' to '.Rbuildignore'
#>  Turn on travis for your repo at https://travis-ci.org/profile/jhrcook
#>  Add a Travis build status badge by adding the following line to your README:
#> Copying code to clipboard:
#>   [![Travis build status](https://travis-ci.org/jhrcook/ExampleAnalysisPackage.svg?branch=master)](h
```

```
usethis::use_appveyor()
#>  Writing 'appveyor.yml'
#>  Adding '^appveyor\\.yml$' to '.Rbuildignore'
#>  Turn on AppVeyor for this repo at https://ci.appveyor.com/projects/new
#>  Add a AppVeyor build status badge by adding the following line to your README:
#> Copying code to clipboard:
#>   [![AppVeyor build status](https://ci.appveyor.com/api/projects/status/github/jhrcook/ExampleAnalys
```

```
usethis::use_coverage("codecov")
#>  Adding 'covr' to Suggests field in DESCRIPTION
#>  Writing 'codecov.yml'
#>   Adding '^codecov\\.yml$' to '.Rbuildignore'
#>   Add a Coverage status badge by adding the following line to your README:
#> Copying code to clipboard:
#>    [![Coverage status](https://codecov.io/gh/jhrcook/ExampleAnalysisPackage/branch/master/graph/badge
#>   Add to '.travis.yml':
#> Copying code to clipboard:
#>    after_success:
#>      - Rscript -e 'covr::codecov()'
```

Just follow the instructions printed out to get everything set up. If this your first time using any of the tools, then you will have to grant them access to your GitHub repositories, and they will do the rest.

The usethis command will also produce the markdown code for showing the status badges for each tools. Placing these below the package name in the README.Rmd is standard practice and will tell pkgdown to put them in the side bar of the site.

On top of looking good and being informative for you during the development process, these badges will also provide visitors an indication as to the quality and maintenance of the package. A few good badges will likely make visitors more trusting of your results.

3.3.3 Spelling

I need not explain why a spell check is useful. Shockingly, this is easy to implement with a usethis function.

```
usethis::use_spell_check()
#>   Adding 'spelling' to Suggests field in DESCRIPTION
#>   Setting Language field in DESCRIPTION to 'en-US'
#> No changes required to /path/to/pkg/ExampleAnalysisPackage/inst/WORDLIST
#> Updated /path/to/pkg/ExampleAnalysisPackage/tests/spelling.R
#>   Run `devtools::check()` to trigger spell check
```

From then on, spelling can easily be checking using the spelling package.

```
# check spelling package-wide
spelling::spell_check_package()
# check spelling in specific files
spelling::spell_check_files("README.Rmd")  # ignores code chunks
```

After making corrections, there are likely still words that are correct but not in the dictionary used by the package. These can be added to a package-specific word-list.

```
spelling::update_wordlist()
```

To skip the confirmation after the above command, use the parameter `confirm = TRUE`.

There is also the option to integrate spelling into the testing process such that it runs automatically anytime the package is checked.

```
spelling::spell_check_setup()
```

3.4 Conclusion

Setting up a R package is actually pretty easy, flexible, and customizable. It is also amendable to change later on (especially because of the usethis package), so there is not need to fret about making it perfect - pieces can always be added later.

Note: At this point, the package is unlikely to pass checks because there are no tests.

Chapter 4

Workflow

4.0.1 Document

```
devtools::document()
```

4.0.2 Test

```
devtools::test()
```

4.0.3 Build and Install

Step one is to build and install the package. This can be done in RStuido with the hot-keys XXX. Otherwise, there is the "Build" button in the "Build" pane. If not using RStudio, then 'devtools' or the command line can be used. the `install()` function *builds* and *installs* the package.

```
devtools::install()
```

If (for some strange reason) only the build *or* install is desired, these two functionalities can be separated.

```
devtools::build()
devtools::install(build = FALSE)
```

At the command line.

```
R CMD build ExampleAnalysisPackage



R CMD INSTALL ExampleAnalysisPackage_0.0.0.9000.tar.gz
```

4.0.4 Check

```
devtools::check()
```

```
R CMD check ExampleAnalysisPackage_0.0.0.9000.tar.gz
```

Chapter 5

Example: Allele-specific ***KRAS*** CNA

[in progress]