R Packages

Thanks to Professor David Gerard

Learning Objectives

- Structure of an R package.
- Documenting R packages.
- Workflow for building R packages.
- Required: Chapters 1–10 and 13–14 from R Packages.
- Resource: Writing R Extensions

Prereqs

- Make sure you have the following packages installed.

```
pkgvec <- c("usethis", "devtools", "roxygen2", "testthat", "knitr", "covr")
for (pkg in pkgvec) {
  if (!requireNamespace(pkg, quietly = TRUE))
    install.packages(pkg)
}
```

- The `{usethis}` and `{devtools}` packages automate many of the tedious tasks of package development, allowing you to focus on writing R code. These are the packages we will mostly use.

Motivation

- Why build an R package?

1. Share your code/methods with others.
2. Re-use functions for yourself.

Package States

- The same R package is in a different format/state at different points of development.
  - Source -> bundled -> binary -> installed -> in-memory.
- **Source package**: A directory of files (R scripts, documentation files, test scripts, etc) with a specific structure. This lecture is about developing source packages.
- **Bundled package**: A source package that has been compressed into a single file (along with a few other operations). These usually end in ".tar.gz". We use the following to create a bundled package from a source package:

```
devtools::build()
```

```
You typically only do this when you are about to submit to CRAN.
```

- **Binary package**: A ready-to-install version for folks who do not have R development tools. You typically don't need to worry about this. If you submit to CRAN, then they will create binaries for you.
- **Installed package**: Installing a package decompresses/places your package in the library directory. This makes it so that you can use `library()` to load a package.
  - Terminology: A **package** is a collection of functions, along with documentation, in a specific format. A **library** is a directory (folder) on your computer that contains installed packages.
  - Confusingly, you use the `library()` function to load a package.
  - You can see/control your active libraries with

```
.libPaths()
```

```
[1] "C:/Users/semiyari/AppData/Local/R/win-library/4.3"
[2] "C:/Program Files/R/R-4.3.2/library"
```

```
- For example, these are some of the packages in C:/Users/semiyari/AppData/Local/R/win-libra
```

```
head(list.files(path = .libPaths()[[1]]))
```

```
[1] "askpass"    "assertthat" "backports"  "base64enc"  "bit"
[6] "bit64"
```

- Ways to install a package:
  - From CRAN: `install.packages()`.
  - From Bioconductor: `BiocManager::install()`.
  - From source package: `devtools::install()`.
  - From GitHub `devtools::install_github()`.
- **In-memory package**: makes functions in a package available for use.
  - Use `library()` to place an installed package in memory.
  - Use `devtools::load_all()` to place a source package in memory. You typically do this during your workflow when you are building your package.

Package Structure

- A typical package will have this directory/file structure

```
.
DESCRIPTION
.git
.gitignore
LICENSE
LICENSE.md
man
    f1.Rd
    f2.Rd
NAMESPACE
R
    rcode.R
.Rbuildignore
README.md
README.Rmd
src
    cppcode.cpp
tests
    testthat
        test-file.R
    testthat.R
```

- Most of these files/folders will be generated by `{devtools}` and `{usethis}`, but you should still know what they are.
- `.git` is a hidden directory that git uses to store your version control history. Don't touch this.

- `.gitignore` is a hidden file used to tell git what files/folders to not place under version control. See the Pro Git Book.
- `LICENSE` and `LICENSE.md` contain the license that your code is distributed under. Typical open-source licenses are MIT and GPL-3.
- The `man` (for "manual") folder contains files that hold your package's documentation. E.g. whenever you use `help()` it uses information from a file in the `man` folder. This package has two functions which are documented `f1()` and `f2()`.
- `NAMESPACE` is a file that determines

  i. What functions are available to the user of your package (versus what functions are for internal use only), and
  ii. What functions from other packages are you importing.

- The `R` folder contains R script files (ending in ".R") that hold all of your R code.
  - R code only goes in R scripts (ending in ".R"), **not** R Markdown Files (ending in ".Rmd").
- `.Rbuildignore` is a hidden file which tells R which files/folders to exclude from the package bundle. You use regular expressions to determine which files to ignore.
  - E.g. if you have a website in your package folder, then you can exclude it from the bundle by placing that folder's name in `.Rbuildignore`.
  - You typically just use `usethis::use_build_ignore()` to add files/folders to `.Rbuildignore`.
- `README.md` is the file that other developers typically first look at, and it is the front page of your package's GitHub website. `README.Rmd` is an R markdown file that generates `README.md`.
- `src` is a folder that contains C++ files (ending in ".cpp").
- `tests` is a folder that contains R code for unit-tests, which are automatic checks that you write to determine if your R package works as you intend.

Create a package skeleton

- You can create a package skeleton with the `usethis::create_package()`.
- Before running this, change your working directory to where you want to create your R package with "Session > Set Working Directory > Choose Directory…".
- This will be the "source" state of the package, so you can choose it to be almost anywhere on your computer.

- Choose a location that is not inside an RStudio project, another R package, another git repo, or inside an R library.
- Then just type

```
usethis::create_package(path = ".")
```

- I don't like RStudio projects, so I typically run

```
usethis::create_package(path = ".", rstudio = FALSE, open = FALSE)
```

```
You can use RStudio Projects if you want. But I won't help with any issues you have with RStu
```

- **Example**: For this lecture, we will create a simple R package called `forloop` that reproduces some Base R functions using for-loops. Create a folder called "forloop", set the working directory to this folder, and run

```
usethis::create_package(path = ".", rstudio = FALSE, open = FALSE)
```

The R folder

- Here, we will discuss how programming is a little different compared to working in an R script or an R Markdown file in interactive mode.
- All R code in package should be a function definition (with **very** few exceptions).

```
fname <- function(arg1 = val1, arg2 = val2, ...) {
  ## code here
  return(retval)
}
```

- Don't have R code outside of a function definition in your package until you **really** understand the benefits of exceptions to this rule.
- All R code should go in R scripts (ending in ".R") **not** R markdowns (ending in ".Rmd")
- Use informative file names. Put only related R functions into the same file (e.g. a main function and some helpers).
- As you add or modify function definitions, you should test interactively test them. That is, iteratively:

1. Use `devtools::load_all()` to load a source package into memory.
2. Play the function you are working on, edit it.
3. Repeat 1 and 2 until you are happy with the function.

- In a typical R script (outside of an R package), code is run when you run it. In an R package, code is run when the package is built. So, for example, if you include the following line of code in your package.

```
x <- Sys.time()
x
```

```
[1] "2024-02-20 23:14:13 EST"
```

```
Then `x` be the time of the package build. If you want the time that a user runs some code, 
```

```
ftime <- function() {
  return(Sys.time())
}
ftime()
```

```
[1] "2024-02-20 23:14:13 EST"
```

- When you alias a function from another package, don't do

  ::: {.cell layout-align="center"}

  ```
  foo <- pkg::bar
  ```

  :::

  instead, do

  ::: {.cell layout-align="center"}

  ```
  foo <- function(...) pkg::bar(...)
  ```

  :::

  This is since `foo` is defined as `pkg::bar` during build time of your package. So if the `{pkg}` maintainers fix an issue in `bar()`, your aliased function will still be the incorrect version of `bar()` until a user rebuilds your package.

- Don't modify a user's R landscape (the global settings and the behavior of functions/objects outside of your package). With rare exceptions, here are some things to not do:
  - Never use `setwd()`.
  - Never use `library()` or `require()`.
    - See below for using other packages in your package.
  - Never use `source()`.
    - Use `devtools::load_all()` while developing a package (but *never* have `devtools::load_all()` in your package).
  - Never change the options via `options()` or `par()`.
  - Never use `set.seed()` to alter the random number generation for a user.
    - Except possibly in examples, vignettes, and unit tests. But never in anything in the /R folder.
  - Never use `Sys.setenv()` or `Sys.setlocale()`.
- **Example**: Let's work together to build a function called `col_means()` that will take as input a data frame and return a vector of column means. We will not use the `colMeans()` function.
- **Exercise**: Create an R script file in your package called "sum.R" via

```
usethis::use_r(name = "sum")
```

```
In this file, create a function called `sum2()` that takes as input a numeric vector `x` and
```

- **Exercise**: Include an `na.rm` argument that defaults to `FALSE`. It removes `NA`'s if `TRUE` and does not if `FALSE`.
- **Exercise**: Create a function called `count_na()` that will use a for-loop to count how many `NA`'s there are in a vector.
- **Exercise**: There are a couple edge cases you should worry about. If the length of `x` is 0, then you should return `NA_real_`. If all values of `x` are `NA`, then you should return `NA_real_` (use `count_na()` to check for this). Edit your function to make these changes now. Test it out on

```
sum2(c(NA, NA, 1), na.rm = TRUE) ## should be 1
sum2(c(), na.rm = TRUE) ## should be NA
sum2(c(NA, NA, NA), na.rm = TRUE) ## should be NA
```

Documentation

- Documentation: Describing:

  1. What a function does.
  2. What are the inputs of the function.
  3. What are the outputs of the function.
  4. Example usage of the function.

- Documentation is vital for

  1. Maintaining packages (you will forget what your functions do)
  2. Having other folks use your package (they need a way to learn the functions).

- You should be writing documentation while you are writing R code

  – **Not** only after the code is "done".

- Documentation in an R package is in ".rd" files in the "man" folder. This is rather esoteric, so we'll use `{roxygen2}` to generate them automatically.

- `{roxygen2}` documentation is provided by comments above a function, where each line begins with `#'`.

  ::: {.cell layout-align="center"}

  ```
  #'
  #' Documentation goes here
  #'
  fn <- function() {
    ## Function code here.
  }
  ```

  :::

- After you write some documentation, you can run

  ::: {.cell layout-align="center"}

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

  :::

  and `{roxygen2}` will automatically update your documentation.

- You can then look at your documentation by using `?` or `help()`.

- `{roxygen2}` comments are formatted as tag-value pairs, where tags begin with an ampersand `@`.
- Values of a tag extend from the tag to the next tag.
- A typical `{roxygen2}` documentation looks like this

  ::: {.cell layout-align="center"}

```
#' @title One line description of what the function does.
#'
#' @description One paragraph description of what the function does
#'
#' @details
#' Long documentation, detailing exactly what the function does
#'
#' @param arg1 What is arg1?
#' @param arg2 What is arg2?
#'
#' @return What is returned?
#'
#' @author Your name
#'
#' @examples
#' ## Some example code goes here
fn <- function(arg1, arg2) {
  ## Function code here
}
```

  :::

- `@param`: Each argument should be documented. You should state

  1. What is the format of the argument (character vector? data frame? numeric matrix?)
  2. What affect the argument has on the function's behavior.

- `@examples`: Include a few lines of example R code. Do not use `@example` as this expects only one line.
- `@return`: What does your function return (numeric vector, character matrix, etc). Describe not just its type but what it is (posterior probabilities, summation, geometric means, etc)
- Use `@inheritParams` to use the parameter documentation from a function in a different function.