# An introduction to creating R packages

Tom Wilson thomas.wilson@gov.scot

5 October 2022

# What is an R Package?

- A package is what we may install from CRAN (or an internal server) with `install.packages()` and load with `library()`.
- Bundles R code within functions so its usable by others after they install and load the package.
- A package can include documentation, sample data, test scripts.
- It is surprisingly easy to create an R package and highly recommended!

# Why create an R Package?

- Sharing: Make processes you have developed easily accessible to others by installing the package, not copying and pasting code or script files.
- Reusability: Packages will usually bundle functionality that may be used across many projects.
- Reproducibility: Custom R packages may be used in, or developed to deliver a Reproducible Analytical Pipeline.
- Clarity: R packages all follow a similar format in where R code is stored, how documentation is created, how test scripts are written. Instead of each person inventing their own way.
- Knowledge: Developing an R package can give a better understanding of how R works.

# An introduction to creating R Packages

## Aim to cover in this talk

1. Why create a package? (already covered)
2. Useful reference materials and useful R libraries.
3. Basic run through of creating a package in R Studio.
4. Documenting functions within a package using `roxygen2`.
5. Checking a package for errors.
6. Building / compiling the package.

# An introduction to creating R Packages

## This introductory talk won't cover

1. Use of Git and GitHub.
2. Formal code testing.
3. Creating vignettes.
4. Releasing a package on CRAN.
5. Rewriting R code in C++ and using Rcpp for performance.

# Reference materials

- Creating R Packages by Hadley Wickham and Jenny Bryan. Easy to follow introductory guide, based around devtools and related libraries.
- Writing R Extensions. Definitive guide on CRAN for reference.

# R Packages that are useful for creating a package

- devtools a bit like a 'tidyverse' for package development.
- usethis automating package setup and configuration tasks.
- roxygen2 documentation and management of package metadata files.
- testthat formal unit testing of packages.

# Creating a package

```
usethis::create_package("rpackagetest")
```

The package name must:

- Be at least two characters
- Letters, numbers and . only (no other chracters including _ -)
- Start with a letter
- Can’t end with a .

```
# Might use available package to check package name
available::available("mypackagename")
```

# Structure of an R package

- DESCRIPTION file provides important information about the package. The title, version, authors, description can be edited. Imported packages will later be added to this using `usethis::use_package("package_name")`.
- NAMESPACE contains the functions that will be available to users of the package.
- `R\` folder contains the code that will be created.

# Make package a git repository

```
usethis::use_git()
```

Does the same as `git init` but adds certain files to `.gitignore`.
If want to put package on GitHub:
- Easiest might be to create GitHub repo first of name of package.
- Then can set the remote:

`git remote add origin <github repo ssh link>`

Alternatively, the `usethis` package does have a `use_github` function might explore.
(Beware of main vs master if using older usethis package version).

# Create first function

```
usethis::use_r(clean_column_names.R)
```

- This simply creates a script to create a function under `R/`.
- Often see each main function made available to uses of a package will be in its own R script under `R/`.

# What should go in the R/.R files

- Note that only function definitions should be in these files.
- Do not use `library(package_name)` to load other libraries.
- Do not use `source("my_other_script.R")`.
- It is fine to have more than one function in one .R file, generally one main function and its helper function.
- See chapter 7 of Building R Packages for a discussion on these things.

# Create a documentation for the function.

In RStudio with cursor inside the function go to menu and select Code > Insert roxygen skeleton.
Will see the roxygen comment using the `#'` combination with distinct sections defined by `@`.

```
#' Title
#'
#' @param df
#'
#' @return
#' @export
#'
#' @examples
standard_scale_df <- function(df) {
  df %>% dplyr::mutate_if(is.numeric, std_scale)
}
```

# Document a function - roxygen2

- The roxygen function comment generates the help for the function.
- Need to complete the template generated by Code > Insert roxygen skeleton:
  - A title describing the function in one line at the top.
  - If leave a blank line after the title sentence can then add more descriptive text until the first @.
  - A description of each input argument next to `@param`.
  - What it returns next to `@returns`.
  - An example of using it below the `@examples`.
- The `@export` means the function will be available to users once package loaded (adds to NAMESPACE). Don't write anything next to this.

# Importing other packages used in functions

- In most cases best to specify packages as `imports` not `depends`.
- If use `imports` then package is used within custom function like `dplyr::mutate`.
- If use `depends` then specified package is attached everytime.
- Helper function that adds package to `DESCRIPTION` > `imports`

```
usethis::use_package("dplyr")
```

# Generate the documentation from roxygen comments

After populating the roxygen2 comments for exported functions, can generate the documentation.

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

- Creates a `man/<function_name>.rd` documentation file
- Uses the `@export` tag to add functions that will be available to users in the NAMESPACE file.

# Load the package to test the functionality

```
devtools::load_all()
```

- This is simulating installing the package and loading it, so useful for checking while developing.
- Note the functions in the package will be available, but not seen in the global environment, just like when load with `library()`.

# Suggestions

- `devtools::check()` is a good way to ensure everything is in order.
- If need to use `%>%` or `:=` specify in roxygen function comment with `@importFrom`

```
#' @importFrom rlang :=
#' @importFrom magrittr %>%
```

- `usethis::use_pipe()` is an alternative way, setting a package so can use magrittr pipe `%>%` in every function.
  (definitely should not have `library(magrittr)` in package R files).

# Distributing - building a package

- Use `devtools::build()`.
- If want a binary use `devtools::build(binary=TRUE)`.
- People can also install package from GitHub `devtools::install_github()`.
- The `.Rbuildignore` specifies files that will not be included in package tarball or binary.
- Note that might wish to increment the version before building. This is in DESCRIPTION.
- However, there is a `usethis` function to do this:

```
usethis::use_version() #prompts with options of how to increment the version number.
```