Working with 

A University of Queensland Advanced Workshop

Session 13:
R Packages for Neophytes

Bill Venables, CSIRO/Data61, Dutton Park

Rhetta Chappell, Griffith University

2–5 February, 2021

Contents

1 Preliminaries

1.1 Why packages?

- Code organisation and discipline.
- Portability.
- In the case of *Rcpp* packages especially, makes the use of compiled code simple, portable and transparent.
- Cooperation with other R users.
- Inclusion on CRAN and recognition.
- Focus for citation to your work after *successful* submission to CRAN or bioconductor.

1.2 Where should packages be installed?

- A new package starts out as a *working directory* with a special structure. On **RStudio** this will be a *package project*.
- When a package is ready, it is *built* into a *source* package, which is a single compressed file containing the *package project*, but with some additions and modifications. This is the "tape archive" file, such as `WWRCourse_0.2.3.tar.gz` that we used for the course materials.
- A source package may alternatively be built as a *binary package*, which is now specific to the operating system.

  On **Windows**, a binary version of our course package would be called `WWRCourse_0.2.3.zip`. This then allows faster installation, does not require the extra software tools needed to build it, but is specific to the operating system.

- The process of *installing* a package means expanding it, ejecting all unnecessary components, compiling all code into fast loading byte-compiled versions and placing it somewhere where the working **R** session can find it. But where?
  The function *.libPaths()* shows you the *package* search path, (as *search()* does the *object* search path):

```
.libPaths()
```

```
[1] "/home/bill/R/x86_64-pc-linux-gnu-library/4.0"
[2] "/usr/local/lib/R/site-library"
[3] "/usr/lib/R/site-library"
[4] "/usr/lib/R/library"
```

  The first entry should be in your home folder, normally a folder like the first above. You may need to make it to allow you to install packages locally, and not interfere with the **R** home library area. (This also has big advantages when you update **R** itself.)

- If you do not have a local package folder, (and *.libPaths()* will offer some clue), you should consider making one, but cautiously!
  On **Windows**, for example, you would use something like:

```
if(!dir.exists("~/R/win-library/4.0")) {
  if(dir.create("~/R/win-library/4.0", recursive = TRUE)) {
    cat("The folder '"~/R/win-library/4.0"' successfully created\n")
  } else {
    cat("The folder '"~/R/win-library/4.0"' cannot be created!\n")
  }
} else {
  cat("The folder '"~/R/win-library/4.0"' already exists\n")
}
```

- On other operating systems the `win-` part will change to something else, such as `x86_64-pc-linux-gnu-` on my home machine, and the final component, `4.0`, changes with the version of **R** changes.

1.3 Tools

Simple packages not involving compiled code may be made using only the basic tools provided by **R** itself. We see an example later.

For packages involving compiled code, extra compiler tools are needed, but these are available in convenient forms for **Windows** and **Mac OS X**, (and for **Linux** they are generally part of the operating system suite, anyway).

Details of the extra tools are summarised in the preliminary sheet sent out before the start of the workshop.

1.4 Structure of a package

- An *installed* R package is a *directory* (or *folder*) in your file system with a number of mandatory components:
  - Two text files named `DESCRIPTION` and `NAMESPACE` which contain context information and directives for the package, and
  - Two *sub-directories* called `man` (for help information) and `R` (for R code files or scripts).

  There will always be a number of other files and sub-directories such as `data` (for data files), `src` (for compiled code) and `doc` (for other information such as package vignettes)

- To *build* an **R** package you create another directory on your file system that is very similar to this structure. There are two common ways to do this with supportive tools:
  - From within **R** itself using functions
    * *package.skeleton* for simple packages or
    * *Rcpp::Rcpp.package.skeleton* for packages with compiled code in **C++** using the *Rcpp* connector package.
  - From within **RStudio** using support packages *devtools* and usually *roxygen2*.

  The first approach has the advantage of allowing the package author more control over the process, but requires more work and can become very tedious. The student is left to explore.

  The second approach, which we illustrate here, is simpler, requires less work and it is easier to maintain or modify the package. The drawback is that in assigning so much responsibility to software it is easy to become stuck if anything goes wrong!

1.5 Automating documentation with `roxygen2`

- The idea is that information on *how to construct*
  - the help files (*.Rd* files), as well as
  - the NAMESPACE file if desired,

  is included *in template form* in the **R** scripts and/or source files as *structured comments*.
- Using *roxygen* comments is another thing to learn, but easy, and the benefits are
  - code and concomitant information are all held in one file which is the only file needing to be changed if the code changes, and
  - the tedious and error-prone task of constructing and maintaining individual and separated *.Rd* files by hand is avoided.

Example:

```
gcd <- function(a, b) {
  if(b == 0) a else Recall(b, a %% b)
}
```

The **RStudio** *Code* menu allows a template *roxygen* set of comments to be automatically generated. With the cursor at the head of the function this generates:

```
#' Title
#'
#' @param a
#' @param b
#'
#' @return
#' @export
#'
#' @examples
gcd <- function(a, b) {
  if(b == 0) a else Recall(b, a %% b)
}
```

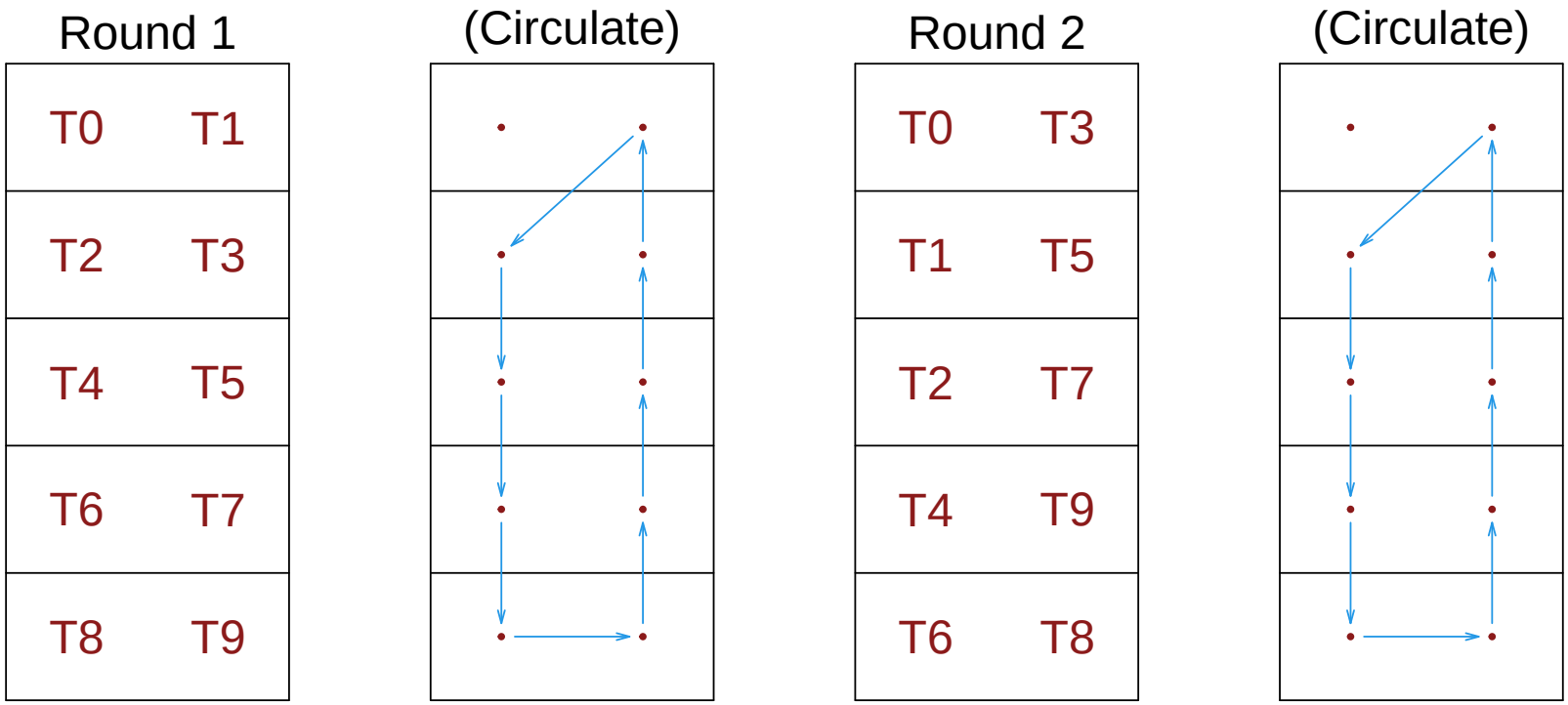
The programmer then simply has to fill in the details:

```r
#' Euclidean algorithm
#'
#' Find the greatest common divisor of two integers
#'
#' @param a A single integer value
#' @param b A single ingeter value
#'
#' @return An integer value the GCD of \code{a} and \code{b}
#' @export
#'
#' @examples
#' gcd(24, 36)  ## answer: 12
gcd <- function(a, b) {
  if(b == 0) a else Recall(b, a %% b)
}
```

and the code is ready for inclusion in a package. The keyboard shortcut for this is by default Ctrl-Alt-Shift-R, (what else?), but may be changed. I have chosen to change it to F12.

2 Example 1: Make Round Robin Tournaments

There is a neat algorithm for doing this sometimes tricky task:



How can we implement this in **R**? We'll make a package for re-use.

Case 1: The no-frills version

A basic code for a function to make a tournament could be:

```
tournament <- function(teams) {
  if(length(teams) %% 2 == 1) { ## odd number of teams -> byes needed
    teams <- c("<Bye>", teams)
  }
  n <- length(teams)
  tournament <- vector(length = n-1, mode = "list")
  for(j in 1:(n-1)) {
    tournament[[j]] <- matrix(teams, ncol = 2)
    teams <- teams[c(1, n/2+1, if(n/2 > 2) 2:(n/2-1), ## else NULL
                     (n/2+2):n, n/2)]
  }
  return(tournament)
}
```

To see what it does:

```
tournament(LETTERS[1:5]) ## Five (anonymous) teams!

[[1]]
     [,1]    [,2]
[1,] "<Bye>" "C"
[2,] "A"     "D"
[3,] "B"     "E"

[[2]]
     [,1]    [,2]
[1,] "<Bye>" "D"
[2,] "C"     "E"
....
     [,1]    [,2]
[1,] "<Bye>" "B"
[2,] "E"     "A"
[3,] "D"     "C"

[[5]]
     [,1]    [,2]
[1,] "<Bye>" "A"
[2,] "B"     "C"
[3,] "E"     "D"
```

The file *PackageMaterials/Example_2/round_rob.R* has a slightly more swish version of this function together with one extra function to provide a schedule, showing for each team which competitor they face in each of the rounds.

Another important feature of this script is how it gives *class* attributes to the objects and provides *print* methods that deliver neater printed displays.

We will now use this code to make a simple package. The steps are as follows:

- Start by making a special folder for the new *package* projects we are about to make. I will assume this is *./Temp*, that is a sub-folder of the workshop project folder.
- File ⟶ New Project ⟶ New directory ⟶ **R** Package
- Give the package a name (e.g. *roundRob*).

  NOTE: package name rules are similar to **R** object name rules, but the *may not* contain underscores. **RStudio** will "pink out" package names that violate these rules.
- Add any code files using the `Add` button. (i.e. `./PackageMaterials/Example_2/round_rob.R`, but also add the two data files `NRL.rda` and `AFWL.rda` from the same folder.)
- Next move to the box "*Create a project that is a subdirectory of:*" and *Browse* to select the `Temp` subfolder you have just created.
- Click on `Create Project` and stand well back.

A new **RStudio** session will start *inside* the **R** package being constructed. At this stage all you have is a *package project*.

To *build* the package you need to do one more step:

- Go to the *Build* tab and select *Install and Restart*.

This will make an (embryonic) package and *install* it in your default library (first position on *.libPaths()*). **R** will restart and the initial command

```
library(roundRob)
```

will automatically be issued.

Moreover your package will also contain a `data` subfolder with your two data items. One way to try out your new package would be

```
tournament(NRL)
```

which should show you a potential preliminary round tournament for the NRL.

There are *print* methods provided, but as these have not been *registered* as such, they will not work as intended. You can, however, use the ungainly

```
print.tournament(tournament(AFLW))
```

to see how the result should look. We now see how we can do this, so clean up at this point and we will look at a more developed package.

- Use *detach(package:roundRob)* to detach it from your **R** session.
- Uninstall this version of the package using *remove.packages("roundRob")*
- Exit from your package project by *File* $\longrightarrow$ *Open project* and re-start your workshop project again.
- Delete the package project folder as well. Do this with some care:

```
dir("./Temp")  ## check that it's where you think it is
unlink("./Temp/roundRob", recursive = TRUE)  # To delete it
```

And we'll start again.

2.1 Adding documentation and a better NAMESPACE

Now re-construct the package using `./PackageMaterials/Example_2/round_rob_rox.R` rather than the previous R script. This one has the *roxygen2* comments inserted.

- Go through the previous process to set up the package project, but do not as yet build the package.
- Look over the *roxygen2* comments in the R script and see if you can understand what they are doing.
- Edit the `DESCRIPTION` file to complete all fields as indicated. At least add a newline at the end of the file.
- Go to the *More* button of the package *Build* tab and select *Configure package build tools*. Tick the last box to ensure *roxygen* is invoked to produce the `.Rd` files and modify the `NAMESPACE` file.
- Delete the temporary `NAMESPACE` file. *file.remove("NAMESPACE")*.
- Click on *Build* $\longrightarrow$ *Install and Restart* and cross your fingers!

If things have worked as they should, you should now be in a new R session with your newly created and installed package attached to the search path. Check to see that the functions work.

```
(tourn <- tornament(AFLW))   ## gives a full torunament
team_schedule(tourn)         ## unpicks the tournament for each team
```

2.2 Source and binary package files

Once the package has been built, installed and checked, you should then build a *source* package (as a `.tar.gz` file) that allows the built package to be transported onto other machines or operating systems.

In addition, you may want to create a *binary* package file that allows your package to be installed on other machines with the same operating system easily. On **Windows** this will be a `.zip` file, for example.

Both of these operations are available through the *More* menu of the package *Build* tab.

NOTE: Windows binary packages are usually built for both 32- and `64-bit` platforms. This is *not* done by default in **RStudio**. If you wish to do this you will need to change a few options, or build the package outside **RStudio**.

2.3 Adding a vignette

Now things get interesting.

- Consider adding a small vignette saying what is going on. At the present time this is done somewhat indirectly:

```
library(devtools)
use_vignette("planning_a_tournament")
```

- After editing your tentative vignette, make sure your build tools also covers building vignettes. Alternatively, or as well, build them directly using

```
build_vignettes()
```

- Complete the package build and install as usual, and try it out.

3 Example 2: A package with R and C++ code

This is definitely more tricky however you do it, but easy enough once you have done it a few times.

The `PackageMaterials/example_cpp` subdirectory contains a file `localWeighting.cpp` containing the **C++** code for a simple local smoother and a kernel density estimate, and the **R** script `local_smooth.R` contains linking functions, together with the *roxygen2* comments

```
dir("./PackageMaterials/example_cpp/")
```

```
[1] "local_smooth.R"     "localWeighting.cpp"
```

Rather than set out in detail all the steps as we did above, the process will be demonstrated during the session. Here are a few preliminary general notes.

To get anywhere you will need to have the package building tools needed already installed on your machine. For **Windows** this means you will need to have the *Rtools* bundle installed and ideally *MiKTeX* as well. This is the major issue.

- Start the build process in much the same way as before, however
  - Choose a *Package* with a new directory, rather than choosing the option *Package using Rcpp*.
  - At the next window begin with *Package type* and choose *Package w/Rcpp*.
- With the package project set up and the package waiting to be built
- Modify the DESCRIPTION file as before, but *DO NOT REMOVE* the NAMESPACE file yet!
- Adjust the build tools to include using *roxygen* in the same way as before, *but do not include generating a NAMESPACE* yet.

- The file `R/local_smooth.R` starts with a few *roxygen* comments

```
#' @useDynLib miscSmooth
#' @import Rcpp
#' @import stats
NULL
```

  Make sure the name on the first line matches the name you have given for your package *exactly*. This is really important – as is everything else!

- In the console window type the command:

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

  This will do an initial building of the **C++** code, but will not yet change the `NAMESPACE`.

- Now do three things
  - adjust the bulid tools so that a NAMESPACE file *is* constructed, (tick *all* the boxes),
  - Remove the NAMESPACE file,
  - Click on *Install and Restart*, cross your fingers and stand well back!

The process becomes more straightforward with practice, of course.

The reason for this slightly indirect way of going about thins is that the *roxygen* functions, by default, will not change a NAMESPACE file that it has not itself created.

But before it can create a new NAMESPACE file the function *Rcpp::compileAttributes* needs to be invoked to know what to include on the new NAMESPACE file.

This latter function does *complieAttributes()*, *does* require at least a nominal NAMESPACE to be in place for it to work.

The second call to *document()* (indirectly, via *Install and Restart*)

does not need to call *complieAttributes* again, though, so it can now write the new NAMESPACE file using information that has already been generated.

There are more elegant ways around this but all require some level of hand-holding if you are to rely on the software available.

Session information

Date: 2021-01-29

- R version 4.0.3 (2020-10-10), `x86_64-pc-linux-gnu`
- Running under: `Ubuntu 20.04.1 LTS`
- Matrix products: default
- BLAS: `/usr/lib/x86_64-linux-gnu/blas/libblas.so.3.9.0`
- LAPACK: `/usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.9.0`
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: dplyr 1.0.3, english 1.2-5, forcats 0.5.1, ggplot2 3.3.3, ggthemes 4.2.4, gridExtra 2.3, knitr 1.31, lattice 0.20-41, patchwork 1.1.1, purrr 0.3.4, readr 1.4.0, scales 1.1.1, stringr 1.4.0, tibble 3.0.5, tidyr 1.1.2, tidyverse 1.3.0, WWRCourse 0.2.3, WWRData 0.1.0, WWRGraphics 0.1.2, WWRUtilities 0.1.2, xtable 1.8-4
- Loaded via a namespace (and not attached): assertthat 0.2.1, backports 1.2.1, broom 0.7.3, cellranger 1.1.0, cli 2.2.0, colorspace 2.0-0, compiler 4.0.3, crayon 1.3.4, DBI 1.1.1, dbplyr 2.0.0, ellipsis 0.3.1, evaluate 0.14, fansi 0.4.2, fractional 0.1.3, fs 1.5.0, generics 0.1.0, glue 1.4.2, grid 4.0.3, gtable 0.3.0, haven 2.3.1, highr 0.8, hms 1.0.0, httr 1.4.2, iterators 1.0.13, jsonlite 1.7.2, lazyData 1.1.0, lifecycle 0.2.0, lubridate 1.7.9.2, magrittr 2.0.1, MASS 7.3-53, modelr 0.1.8, munsell 0.5.0, parallel 4.0.3, PBSmapping 2.73.0, pillar 1.4.7, pingr 2.0.1, pkgconfig 2.0.3, processx 3.4.5, ps 1.5.0, R6 2.5.0, randomForest 4.6-14, Rcpp 1.0.6, readxl 1.3.1, reprex 1.0.0, rlang 0.4.10, rpart 4.1-15, rstudioapi 0.13, rvest 0.3.6, searchpath 0.1.1, SOAR 0.99-11, splines 4.0.3, stringi 1.5.3, tidyselect 1.1.0, tools 4.0.3, vctrs 0.3.6, withr 2.4.1, xfun 0.20, xml2 1.3.2