Packages

Dom Owens

08/10/2019

Projects for organising code

R has inbuilt functionality for self-contained projects. These are simply directories of files. One standard structure goes as follows:

- "R" folder containing scripts
- "README.Rmd" with a plain-language description of the project
- "data" folder with data, perhaps split into "raw" and "processed"
- "doc" folder with documentation for external use
- "output" folder with reproducible results of applying functions to data

Perhaps other folders can be included, such as "src/" for C++ code.

Code can be loaded from external sources using the "source" or "library" commands

```
#source("x.Rmd")
library(ggplot2)
```

Version control using git and GitHub

Version control software tracks changes in projects, allowing multiple users to work on the same files or previous versions to be recovered.

git is a useful tool for this, and is integrated into RStudio. "Staging" refers to submitting files to the current branch, and can be performed via the git tab in the top right panel. When changes are made, a "commit" should be made; it is good practice to use a single commit for changes to specific parts - this allows easy reversion and the ability to describe changes in plain language. A "branch" can be added to change code without affecting the master version of the code, and then later merged. Files to be ignored for staging should be specified in the ".gitignore" file.

GitHub is a useful platform for displaying and sharing projects. Other users can fork a public repository, edit it and submit a pull request.

R Packages

Packages are collections of functions and data. These have a description and license attached. Documentation can be generated automatically with "Build -> Configure Build Tools... -> Generate documentation with Roxygen" to describe functionality.

Here we see the contents of the Example package, which is a function complete with Roxygen documentation:

```
#' Title
#' Example package
#' @param x
#'
#' @return
#' SUms entries of vector
```

```
#' @export
#'sum
#' @examples
sum <- function(x) {
  s <-0
  for (i in 1:length(x)) {
    s <- s + x[i]
  }
  s
}
```

And here we install and use the package:

```
install.packages("/home/do16317/Documents/Stat Comp 1/Packages/Example", repos = NULL, type = "source")
```

```
## Installing package into '/home/do16317/R/x86_64-pc-linux-gnu-library/3.4'
## (as 'lib' is unspecified)
```

```
Example::sum(1:10) # use sum function from package
```

```
## [1] 55
```

See https://github.com/Dom-Owens-UoB/StatsComp1 for a managed git repository including an installable version of the Example package.

For stability purposes, testing can be applied to package functions. We do this by providing criteria which the functions must meet, in the form of a test suite. The "testthat" package permits this with tests of the form

```
library(testthat)
test_that("sumtest", {
  expect_equal(Example::sum(1:10), 55)
})
```

Sometimes untested code can contain bugs and errors. Automated tools such as "covr" can be used to check the coverage of the code, reporting which lines have or have not run. We use the command

```
#covr::report()
```

Testing with Travis

Packages available on gitub can be tested with travis-ci.org. Continuous integration (CI) allows pull requests to trigger tests, so the owner of the main branch can ensure the new code is working as intended. A ".travis.yml" file should be added to the repository of the R package to allow Travis access. The following code will run tests from "testthat":

```
language: r
dist: xenial
cache: packages
branches:
  only:
    - master

r_github_packages:
  - r-lib/covr

after_success:
```

```
  - Rscript -e 'covr::codecov()'
```

Hosting pages with GitHub pages

Travis can be used to sync web pages to GitHub.

https://dom-owens-uob.github.io/StatsComp1/