Portfolio 2

Sam Bowyer

Organising Code

In the previous portfolio we discussed literate programming and the ways in which it can aid the readability and organisation of code. It is natural, then, to extend this desire for organisation to the ways in which we organise projects made from multiple files (many of which may not just be pure code).

One of the big benefits of thoughtful organisation is that it makes bugs and mistakes (as well as general improvements and alterations) easier to find and correct/implement. For this reason we will also consider version control through *git* and consider the importance of thorough testing practices.

Projects

Although different projects will require different structures, the following is a typical `R` project structure (note that not all projects will require all of these folders):

```
project/
|
|-- R/            # Contains the R scipts of the project (usually not in subdirectories).
|-- README.md     # A short file in markdown describing the project.
|-- data/         # Contains the data needed for the project. Might contain 'raw' and
|                   'processed' subdirectories for the data before and after processing
|                   (this processing should be done by code viewable inside the project).
|-- doc/          # Contains documentation of the project.
|-- output/       # Contains anything generated by the R files. Might contain a
|                   subdirectory for figures/tables/graphs etc.
|-- tests/        # Contains test scripts used to check the validity of the project's
|                   code.
|-- src/          # If the project requires any C/C++ code (usually to improve
                    performance in R using the Rcpp package) then it would go in here.
```

To load the functionality of a single `R` script into a new one, we simply write:

```
source("R/script.R")
```

To load a whole package we use the `library` command, e.g.:

```
library(Rccp)
```

The command '`require`' works very similarly to `library`, except if the package doesn't load (e.g. if the user doesn't have it installed) the former gives a warning and returns `FALSE` before carrying on with program execution whereas the latter gives an error during the package loading time (which, here, we catch). To see this difference, note that `"Success"` is printed after the first codeblock which uses `require`, but is not printed by the second codeblock as the error stops us from moving on to the `cat("Success")` line.

```
requireTest <- function(){
  bool = require(doesntexist)
  if (!bool) {
    cat("Success")
```

```
  }
}
requireTest()
```

```
## Loading required package: doesntexist
```

```
## Warning in library(package, lib.loc = lib.loc, character.only = TRUE,
## logical.return = TRUE, : there is no package called 'doesntexist'
```

```
## Success
```

```
libraryTest <- function(){
  tryCatch({
    library(doesntexist)
    cat("Success")},
  error=function(e){paste(e)})
}
libraryTest()
```

```
## [1] "Error in library(doesntexist): there is no package called 'doesntexist'\n"
```

(N.B.: We've put these tests into functions to help with the RMarkdown output.)

Packages

RStudio has an option to create a package layout via the menu options:

`File -> New Project -> New Directory -> R Package`

This will create the following project structure (along with a git repository if you select the option in the package creation menu).

```
package/
|
|-- DESCRIPTION   # A file describing the package in a standardised format (with fields
|                   such as 'Title', 'Version', 'Author', 'Maintainer', 'License' etc.).
|-- NAMESPACE     # A file denoting which functions of the package should be exported and
|                   made available to users, and which functions from other packages are
|                   needed for this package.
|-- R/            # The R scripts.
|-- man/          # Manuals for the scripts.
```

A more detailed look into the formatting standards of `DESCRIPTION` and `NAMESPACE` can be found here. Although not provided by default, it might also be useful to include some of the other folders described in the previous section too, depending on the nature of the package.

One important part of the `DESCRIPTION` file is the `License` field—in here you would specify the type of copyright license that you are providing this software package with (typically you'd also add the full text of the license to your project in a '`LICENSE`' or '`LICENSE.txt`' file, and maybe a '`LICENSE.md`' file too in order to improve clarity on services like Github). Many different pre-existing licenses exist for you to choose from, such as the MIT license or GNU GPLv2/3 licenses, each describing slightly different restrictions on how other people can use, modify and redistribute your code. Github's choosealicense.com website gives advice on which license you might want to use for a given project, however, it is important to note that there is some ambiguity around the legal validity of some of these licenses—consulting an expert would be the best course of action in order to be completely sure of a project's copyright status.

Git

Version control is an essential part of any project's development, and git provides a much better version control system than naming your files things like `script_v3_(Final)_(IMPROVED).R` and `data_fixed_V2_OLD.dat`. Using git allows you (whether alone or as part of a team) to track the changes in files over time, create multiple `branches` of a project in order to take it in different directions and even merge those branches back together, plus much more with only a few (fairly) simple commands. Although GUIs for git do exist (including one within RStudio), we'll proceed only with the procedures for using git in the terminal as this is the most universal way to use the basic features of git and understand how they works.

With git installed you first set your name and email with the two commands:

```
git config --global user.name "Your Name"
git config --global user.email name@domain
```

(You can check that these values have been set using the command `git config --list --show-origin`.)

You can initialise a git repository by moving to the highest-level folder that you want to track and typing:

```
git init
```

This sets up various (hidden) git files in order to track the folder/repo. Next you need to add the files that you want to track using the `git add <filename>` command. To add all files in the folder (and sub-directories) you would type:

```
git add .
```

(since `.` referes to the current directory). Often there are files in your project that you don't want to track, such as large datasets or cached files. You can specify these files (either specifically or by a filename pattern e.g. `*.csv` to ignore all csv files) one line at a time in the hidden file `.gitignore`—often programs like RStudio will do some of this work for you by automatically adding to this file if it finds out that it's working on files inside a git repo.

So far we've added files to the 'staging area' (which can be seen by typing `git status`), but we still have to `commit` them to the repository to indicate that we want to keep a record of the changes that have been made.

```
git commit -m "A short message describing the changes made in this commit"
```

We can create a new branch of the repo and then start working on it by typing:

```
git branch <branch name>
git checkout <branch name>
```

or, equivalently:

```
git checkout -b <branch name>.
```

Different versions of files throughout commits (and in different branches) can be looked at using the `git checkout` command (or using a service like Github):

```
git checkout <commit hash> -b <branch name>
```

Each commit's hash can be found by typing `git log`.

We can also easily revert a branch back to its last-made commit at any point by typing:

```
git revert HEAD
```

Github

So far we've just considered having local repositories as a version control system, but services like Github allow us to have remote repositories, which make it easier to work on projects with multiple computers and/or people.

Using Github's website we can create a new project with SSH address `git@github.com:sambowyer/testRPackage.git` (we could also use the https address `https://github.com/sambowyer/testRPackage.git` and the rest of each command would stay the same—though SSH requires you to first set up an SSH key with Github). Then we can add our Github repo as a remote location '`origin`', create a '`main`' branch, and push our existing (local) branch to this remote repo's `main` branch using the commands:

```
git remote add origin git@github.com:sambowyer/testRPackage.git
git branch -M main
git push -u origin main
```

Hosting packages on Github also has the benefit that you can install `R` packages directly from Github using:

```
devtools::install_github("https://github.com/sambowyer/testRPackage.git")
```

You can copy a remote repository onto your local branch by typing:

```
git clone <repo address>
```

Once you have a local branch of a repo (with a correct remote location) you can fetch updates from the remote repo:

```
git fetch <remote>
```

And merge those fetched updates into your local branch:

```
git merge <remote>/<branch>
```

Though the `pull` command does both of these in one go:

```
git pull <remote> <branch>
```

If you've cloned someone else's Github repository and worked on your own branch of it (known as a 'fork' of the repo) locally, you can submit a "Pull Request" on their Github repo page to ask for them to merge the main branch with yours (thus incorporating the work you've done on the repo). Typically you would add information justifying your code changes and if a maintainer of the repo approves the request the branches will merge—and hopefully with no conflicts; it's useful to keep pull requests simple enough that conflicts do not arise between branches. The command:

```
git remote add upstream <upstream address>
```

tells your local branch the location of the upstream (i.e. original) branch, so that you can work on your fork whilst making sure it is up to date with the upstream (using `fetch`, `merge`, `pull` etc.).

Testing

It's important that you test the code in your packages to make sure that it works correctly, whether this is through test-driven development (e.g. writing the tests first and then writing the code bit by bit until all of the tests pass) or testing with a less regular schedule.

The `R` package `testthat` provides lots of functionality to test that code does what you expect. First, by running the command

```
usethis::use_testthat(3)
```

the folder `tests` is created in the top-level of the package. This folder contains a file `testthat.R` (which you don't need to modify) and a folder `testthat/` in which you should add test files corresponding to each script in your `R` folder. For instance, if we had a file `R/add.R` containing the following:

```
add <- function(a,b) a+b
```

We might create the file `tests/testthat/add-test.R` containing:

```
test_that("add works", {
  # Multiple `expect_...` functions exist that can be used in a variety of situations
  expect_equal(add(2, 3), 5)
})
```

You can then run the tests in the `testthat/` folder with the command:

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

For example, with just this one test on the (rather uninspired) function above the output would be:

```
  Testing testRPackage
  | F W S  OK | Context
  |          1 | add

  Results
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 1 ]
```

One further analysis we can do of the tests is to use the `covr` package via the command;

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

Which will run the tests and produce a report of which lines in the scripts were actually run during the tests—ideally you would write enough tests so that every line gets run at least once (thus decreasing the chance that any line would produce a bug after testing), however, it's not uncommon for large projects to have lines missed out in testing.

Github Actions

One more benefit of hosting packages on Github is the option to run both testing and coverage reports automatically through Github Actions—an example of continues integration (CI) whereby code changes can be integrated into the larger project through automated tests/coverage checks/style reviews/etc. This not only helps development with multiple parties (e.g. running tests whenever a pull request is made to ensure any new code changes don't stop certain tests passing) but also for solo-development with tests able to run on different versions of `R` or on different operating systems.

Many other Github Actions exist for non-test-related tasks, such as building and publishing webpages (e.g. with Github Pages through this action), running code in a repository at regular intervals, or caching code dependencies to speed up execution of these workflows.