While developing your R package, you will want to make sure the code it contains is as clean as possible and that your package build and testing times are as efficient as you can make them. There are a number of tricks and tools at your disposal to accomplish these aims. This post, the third in a series that covers R package development, will introduce a few of those and demonstrate how they can improve your package development process. We’ll continue with the chifishr .

**lintr**

The lintr package has the ability to scan over your entire package and return messages pointing to “lints”, or areas where the code does not follow certain styling guidelines. This package helps keep your code looking consistent and clean, and can be used in your testing process to ensure new code adheres to the same general rules.

library(lintr)

lintr::lint\_package("chifishr")

We see there are a few lines in our existing code that lintr points out should be cleaned up to adhere to the styling guidelines. To address the “no visible global function definition” notes. This tells the package build process that the %>% function used in our code is defined globally from the dplyr package.

There are also a few aesthetic changes to our testing script that we should clean up: deleting unecessary commented code and keeping the length of our lines within a standard 80 characters. We can also add a test to our test-chi\_fisher\_p.R script that will fail if there are any lints in the package, as seen below.

context("test-chi\_fisher\_p.R")

test\_that("returns chi-squared p value if no warnings are thrown", {

expect\_silent(chisq.test(treatment$gender, treatment$treatment))

expect\_equal(chi\_fisher\_p(treatment, "gender", "treatment"),

chisq.test(treatment$gender, treatment$treatment)$p.value)

})

test\_that("returns fisher p value if chi-squared warnings are thrown", {

expect\_warning(chisq.test(treatment$outcome, treatment$treatment))

expect\_equal(chi\_fisher\_p(treatment, "outcome", "treatment"),

fisher.test(treatment$outcome, treatment$treatment)$p.value)

})

if (requireNamespace("lintr", quietly = TRUE)) {

context("lints")

test\_that("Package Style", {

lintr::expect\_lint\_free()

})

}

Suggests:

testthat,

lintr

**covr**

The covr package tracks test coverage for your R package, going line-by-line to check if each line of code is covered by a test. Measuring code coverage allows developers to run internal quality checks on their code and gives future package users confidence that the package contains high quality code.

# local covr check in RStudio console

library(covr)

covr::package\_coverage("chifishr")

# chifishr Coverage: 100.00%

# R/chi\_fisher\_p.R: 100.00%

Note that covr cannot run during R CMD check. This is because it may modify code and there could possibly be unknown edge cases where that modification affects the output. For that reason, the best practice is to add a line to the .gitlab-ci.yml file that runs covr on your package after the building and testing scripts:

after\_script:

- R -e 'covr::package\_coverage(Sys.getenv("CI\_PROJECT\_DIR"))'

Suggests:

testthat,

lintr,

covr

**styler**

The styler package allows for a high amount of flexibility in regards to configuring the style of the code in your package. It can format complicated expressions that involve line breaking and indention based on both brace expressions and operators. The styler::style\_pkg() function will style the package following tidyverse conventions.

This package is especially useful if you are worried that your code is not styled consistently and may be messy to other users. It can be used during local development to check consistency with spacing and adding new lines in your code and automate the minor changes to code for you. Be aware that running the style\_pkg() function automatically changes your code, so be prepared to review the changes that were made if you choose to take advantage of the styler package.

In our example, let’s set strict = FALSE to set spacing to *at least one* around = signs, keeping them aligned the same way we have written our code.

library(styler)

styler::style\_pkg("~/gitlab/chifishr", strict = FALSE)

# Styling 7 files:

# data-raw/treatment-data.R ✔

# R/chi\_fisher\_p.R ✔

# R/chifishr-package.R ✔

# R/data.R ✔

# R/utils-pipe.R ✔

# tests/testthat.R ✔

# tests/testthat/test-chi\_fisher\_p.R ✔

# ────────────────────────────────────────

# Status Count Legend

# ✔ 7 File unchanged.

# ℹ 0 File changed.

# ✖ 0 Styling threw an error.

# ────────────────────────────────────────

We see that all of the files included in our package are already following tidyverse conventions, so no files were automatically changed by styler. Note that if we had kept strict = TRUE, some files would have been changed due to the spacing around =.

**Optimizing the .gitlab-ci.yml file**

If your package has a high number of dependencies and these need to be downloaded during each build, the build time for the package will be unnecessarily long with the following .yml configuration:

# .gitlab-ci.yml

image: r-base

test:

script:

- R -e 'install.packages(c("dplyr", "purrr", "testthat"))'

- R CMD build . --no-build-vignettes --no-manual

- R CMD check \*tar.gz --no-build-vignettes --no-manual

That leaves a lot of room for improvement! A couple ways we can make this process more efficient are to cache dependencies and use a more advanced docker image.

The following is an improved version:

image: methodsconsultants/r-packaging

variables:

R\_LIBS\_USER: "$CI\_PROJECT\_DIR/ci/lib"

CHECK\_DIR: "$CI\_PROJECT\_DIR/ci/logs"

BUILD\_LOGS\_DIR: "$CI\_PROJECT\_DIR/ci/logs/$CI\_PROJECT\_NAME.Rcheck"

test:

script:

- mkdir -p $R\_LIBS\_USER $BUILD\_LOGS\_DIR

- R -e 'devtools::install\_deps(dep = T, lib = Sys.getenv("R\_LIBS\_USER"))'

- R -e 'devtools::check(check\_dir = Sys.getenv("CHECK\_DIR"))'

- R -e 'if (length(devtools::check\_failures(path = Sys.getenv("BUILD\_LOGS\_DIR"), note = FALSE)) > 0) stop()'

cache:

paths:

- $R\_LIBS\_USER

after\_script:

- R -e 'covr::package\_coverage(Sys.getenv("CI\_PROJECT\_DIR"))'

CI\_PROJECT\_DIR and CI\_PROJECT\_NAME. CI\_PROJECT\_DIR is the full path to where the repository is cloned on the remote machine where the CI job is run, and CI\_PROJECT\_NAME is the project folder name. In the above file we set three new variables, R\_LIBS\_USER, CHECK\_DIR and BUILD\_LOGS\_DIR, which add to these built-in variables and will be referenced in the following script. These will hold the libraries and logs from the CI jobs for caching the dependencies of the package.

The first command in the test script creates a directory, ci, which contains the subdirectories lib and logs. The ci/lib folder will contain the downloaded dependency files and will be cached by the CI servers to be used for future builds. This will eliminate the time needed to download dependencies for your package. The ci/logs folder will contain the logs output from devtools::check, which can be used to verify if any errors or warnings are thrown from the build process.

The second command, devtools::install\_deps(), will install the dependencies for your package *only* if there is not a cached directory containing the dependencies already available. The first time you run this job should be the only time the build needs to download any dependencies, other than if you add new dependencies (only the new dependencies should need to be installed)

Next, the devtools::check command builds and checks your package, with output from the tests saved in the ci/logs directory. The last command in the script then checks for failures in the build logs directory and fails the job if any error or warning messages are thrown.

If no error or warning messages are present, the build passes and the ci/lib directory is then cached on GitLab CI’s server for your project.

**Updating .Rbuildignore**

Since we created the ci folder inside the project prior to the build, we will need to add it to the .Rbuildignore file so that devtools::check() ignores the created folder and all of the contents inside it. Here is the current .Rbuildignore file for chifishr which contains a line for the created ci folder:

^chifishr\.Rproj$

^\.Rproj\.user$

^LICENSE\.md$

^data-raw$

^README\.Rmd$

^\.gitlab-ci\.yml$

^ci