Quince consejos para mejorar nuestro código y flujo de trabajo con R

Francisco Rodríguez-Sánchez

Departamento de Biología Vegetal y Ecología, Universidad de Sevilla, Avda. Reina Mercedes s/n, 41012 Sevilla, España.

Autor para correspondencia: Francisco Rodríguez-Sánchez [f.rodriguez.sanc@gmail.com]

Palabras clave

programming; R; reproducibility

Keywords

programación; R; reproducibilidad

Introducción

La mayoría de los ecólogos que escribimos código informático para desarrollar nuestros análisis somos autodidactas (Hernandez et al., 2012). Nunca hemos recibido formación sobre buenas prácticas de programación o cómo escribir código 'limpio', eficiente y bien estructurado (Wilson et al., 2014, 2017; Rodríguez-Sanchez et al., 2016). Tampoco es frecuente recibir comentarios sobre cómo mejorar un código existente. En consecuencia, nuestro código a menudo es ineficiente, desordenado, proclive a contener errores, difícil de revisar y reutilizar.

En esta nota se recogen 15 breves consejos para mejorar nuestro flujo de trabajo y programación en lenguaje R (R Core Team, 2020). R se ha convertido en la herramienta estadística y lenguaje de programación más popular en ecología (Lai et al., 2019). Estos consejos pretenden evitar errores frecuentes y mejorar la calidad del código desarrollado en nuestros análisis.

1. Utiliza un sistema de control de versiones

En lugar de guardar distintas versiones de nuestro código como `script_v1`, `script_v2`, etc., es muy recomendable utilizar herramientas como `git` que permiten tener un archivo perfectamente organizado de todos los cambios realizados en datos y código. `git` registra minuciosamente quién hizo qué, cuándo y por qué, y permite comparar y recuperar versiones anteriores. Cuando además se

combina `git` con plataformas como GitHub, GitLab o Bitbucket, se facilita enormemente el desarrollo colaborativo de proyectos (Blischak et al., 2016). No obstante, debe tenerse en cuenta que `git` está diseñado para trabajar con archivos no muy grandes y de texto plano (código, datos en formato CSV, etc.).

2. Utiliza una estructura estándar de proyecto

Idealmente, todos los archivos relacionados con un proyecto (datos, código, figuras, etc.) deben alojarse en la misma carpeta (Wilson et al., 2017; Cooper y Hsing, 2017). Esto facilita la organización y la colaboración (al hacer el proyecto fácilmente portable).

3. Añade un fichero README

Añade un fichero README al directorio raíz de tu proyecto que sirva como presentación del mismo (Wilson et al., 2017): objetivos y elementos del proyecto, desarrolladores, licencia de uso, cómo citarlo, etc.

4. Utiliza un script maestro

En proyectos relativamente complejos, donde se manejan varios conjuntos de datos o *scripts* de código, es muy recomendable tener un *script maestro* que se encargue de ejecutar todas las piezas en el orden correcto. Podría ser algo tan sencillo como este `makefile.R`:

```
source("clean_data.R")
source("fit_model.R")
source("generate_report.R")
```

Herramientas como `drake` (Landau, 2018) o `targets` (Landau, 2020) permiten un control mucho más potente del flujo de trabajo, ejecutando solo aquello que necesita actualización, permitiendo paralelizar, etc.

5. Evita guardar el *workspace*

En general, es preferible no guardar el espacio de trabajo (*workspace*, fichero `.RData`) al finalizar cada sesión de trabajo, para evitar la acumulación de objetos innecesarios en memoria. En su lugar, debemos guardar siempre el código fuente, y guardar opcionalmente aquellos objetos (p. ej. usando `saveRDS`) que requieren computación larga o costosa (Bryan y Hester, 2019).

6. Aprovecha las ventajas de Rmarkdown

Rmarkdown (https://rmarkdown.rstudio.com) permite integrar texto y código (no solo de R) y generar documentos dinámicos que reproducen todo el proceso de análisis, incluyendo además todos los

resultados (gráficas, tablas, etc.). Así, Rmarkdown facilita la colaboración y comunicación de resultados, y reduce drásticamente el número de errores (Cooper y Hsing, 2017).

7. Aprovecha las herramientas que ayudan a escribir mejor código

Paquetes como `fertile` (Bertin y Baumer, 2020; Baumer y Bertin, 2020), que nos avisa sobre posibles problemas en nuestro código y formas de solucionarlos, o `Rclean` (Lau, 2020), que nos devuelve el código mínimo empleado para producir cualquier resultado, son muy útiles para escribir mejor código o mejorar código ya existente.

8. Comenta tu código

Utiliza los comentarios para guiar al lector, distinguir subsecciones, o explicar por qué se hacen las cosas de una determinada manera.

9. Utiliza nombres memorables

Utiliza nombres con significado que evoquen/resuman el contenido del objeto (p. ej. `modelo.aditivo`, `modelo.interactivo` en lugar de `m1`, `m2`).

10. Documenta los datos

Prepara metadatos explicando qué representa cada variable (medidas, unidades, etc.), autores, licencia de utilización, etc. Herramientas como `dataspice` (Boettiger et al., 2020) facilitan enormemente esta tarea, y mejoran la visibilización y potencial de reutilización de los datos.

11. Comprueba los datos antes del análisis

En cualquier análisis es bastante probable que los datos de partida contengan errores, introducidos al teclear los datos, importarlos o manipularlos. Por tanto es fundamental comprobar la calidad de los datos antes del análisis. Para ello, paquetes como `assertr` (Fischetti, 2020) o `pointblank` (Iannone y Vargas, 2020) resultan muy útiles. Por ejemplo, el siguiente código

```
library("assertr")

dataset %>%
  assert(within_bounds(0, 0.20), fruit.weight) %>%
  assert(in_set("rojo", "negro"), colour)
```

comprueba que la variable `fruit.weight` contenga valores numéricos entre 0 y 0.20, y que la variable `colour` contenga solo dos valores (“rojo” y “negro”). Si estas condiciones no se cumplen, `assertr` nos avisará.

12. Comprueba los resultados del análisis

Al igual que comprobamos los datos originales, podemos comprobar que los resultados del análisis entran dentro de lo esperado. Por ejemplo, si el resultado debe estar comprendido entre 0 y 1:

```
output %>%
  assert(within_bounds(0, 1), result)
```

Tales comprobaciones son muy útiles para detectar posibles errores en nuestro código, cambios inesperados en paquetes, etc., que producen resultados erróneos.

13. Escribe código modular

Los *scripts* de código largos y desorganizados son más difíciles de revisar y, por tanto, más proclives a contener errores. Es conveniente escribir código modular, por ejemplo, partiendo un *script* largo en varios pequeños (p. ej. `prepare_data.R`, `run_analysis.R`, `make_figures.R`), o escribiendo funciones con un cometido específico e independientes del código principal del análisis.

14. Evita repeticiones

A menudo necesitamos ejecutar unas líneas de código repetidamente. Por ejemplo, para repetir una figura con distintas especies:

```
dataset %>%
  filter(species == "Laurus nobilis") %>%
  ggplot() +
  geom_point(aes(x, y))

dataset %>%
  filter(species == "Laurus azorica") %>%
  ggplot() +
  geom_point(aes(x, y))
```

Estas repeticiones conducen a *scripts* de código largos, difíciles de editar y revisar, y por tanto más proclives a errores.

¿Cómo podemos evitar repetirnos? Una opción podría ser escribir un bucle (`for loop`) con iteraciones para cada especie:

```
species <- c("Laurus nobilis", "Laurus azorica")
for (i in species) {
  dataset %>%
    filter(species == i) %>%
```

```
    ggplot() +
    geom_point(aes(x, y))
}
```

Aún mejor, podríamos escribir una función que produzca la gráfica para una especie dada:

```
plot_species <- function(sp, data) {
  data %>%
    filter(species == sp) %>%
    ggplot() +
    geom_point(aes(x, y))
}
```

Y después ejecutar esa función para todas las especies. Por ejemplo, usando `lapply`:

```
lapply(species, plot_species, data = dataset)
```

o `purrr` (Henry y Wickham, 2020):

```
purrr:::map(species, plot_species, data = dataset)
```

15. Registra las dependencias

Todo análisis depende de un conjunto de paquetes que conviene documentar de manera consistente e interpretable. Ello nos permite, por ejemplo, ejecutar fácilmente el análisis en otro ordenador, o recrear el entorno computacional tras una actualización.

Existen muchas opciones de documentar las dependencias de nuestro análisis, desde la función `sessionInfo`, paquetes como `automagic` (Brokamp, 2019) o `renv` (Ushey, 2020) que guardan un fichero de texto con todos los paquetes utilizados (y sus versiones), a paquetes como `containerit` que facilitan la creación de un 'dockerfile' para recrear el entorno computacional en cualquier computadora (Nüst et al., 2020).

Muchas de estas medidas son fáciles de implementar y no requieren grandes cambios en la organización del trabajo ni el estilo de programación, no obstante pueden contribuir a mejorar notablemente la calidad del código desarrollado para nuestros análisis, redundando por tanto en beneficios tanto para el programador como sus colaboradores y revisores.

146 REFERENCIAS

147 Baumer, B.S., Bertin, A. 2020. *fertile: Creating Optimal Conditions for Reproducibility*.

148 Bertin, A.M., Baumer, B.S. 2020. Creating optimal conditions for reproducible data analysis in R with
149 «fertile». *Stat*.

150 Blischak, J.D., Davenport, E.R., Wilson, G. 2016. A Quick Introduction to Version Control with Git and
151 GitHub Ouellette, F. (ed.), *PLOS Computational Biology* 12: e1004668.

152 Boettiger, C., Chamberlain, S., Fournier, A., Hondula, K., Krystalli, A., Mecum, B., Salmon, M. et al.
153 2020. *dataspice: Create Lightweight Schema.org Descriptions of Data*.

154 Brokamp, C. 2019. *automagic: Automagically Document and Install Packages Necessary to Run R*
155 *Code*.

156 Bryan, J., Hester, J. 2019. *What They Forgot to Teach You About R*.

157 Cooper, N.H., Hsing, P.-Y. eds.. 2017. *A guide to reproducible code in ecology and evolution*. British
158 Ecological Society.

159 Fischetti, T. 2020. *assertr: Assertive Programming for R Analysis Pipelines*.

160 Henry, L., Wickham, H. 2020. *purrr: Functional Programming Tools*.

161 Hernandez, R.R., Mayernik, M.S., Murphy-Mariscal, M.L., Allen, M.F. 2012. Advanced Technologies
162 and Data Management Practices in Environmental Science: Lessons from Academia.
163 *BioScience* 62: 1067-1076.

164 Iannone, R., Vargas, M. 2020. *pointblank: Validation of Local and Remote Data Tables*.

165 Lai, J., Lortie, C.J., Muenchen, R.A., Yang, J., Ma, K. 2019. Evaluating the popularity of R in ecology.
166 *Ecosphere* 10:.

167 Landau, W.M. 2020. *targets: Dynamic Function-Oriented 'Make'-Like Declarative Workflows*.

168 Landau, W.M. 2018. The drake R package: a pipeline toolkit for reproducibility and high-performance
169 computing. *Journal of Open Source Software* 3:.

170 Lau, M. 2020. *Rclean: A Tool for Writing Cleaner, More Transparent Code*.

171 Nüst, D., Sochat, V., Marwick, B., Eglen, S.J., Head, T., Hirst, T., Evans, B.D. 2020. Ten simple rules
172 for writing Dockerfiles for reproducible data science Markel, S. (ed.), *PLOS Computational*
173 *Biology* 16: e1008316.

174 R Core Team. 2020. *R: A Language and Environment for Statistical Computing*. R Foundation for
175 Statistical Computing, Vienna, Austria.

176 Rodríguez-Sanchez, F., P'erez-Luque, A.J., Bartomeus, I., Varela, S. 2016. Ciencia reproducible:
177 qué, por qué, cómo? *Ecosistemas* 25: 83-92.

178 Ushey, K. 2020. *renv: Project Environments*.

179 Wilson, G., Aruliah, D.A., Brown, C.T., Hong, N.P.C., Davis, M., Guy, R.T., Haddock, S.H.D. et al.
180 2014. Best Practices for Scientific Computing Eisen, J. A. (ed.), *PLoS Biology* 12: e1001745.

181 Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., Teal, T.K. 2017. Good enough practices
182 in scientific computing Ouellette, F. (ed.), *PLOS Computational Biology* 13: e1005510.

183 TABLA 1

184 **Tabla 1**. Quince consejos para mejorar nuestro código y flujo de trabajo en R.

1. Utiliza un sistema de control de versiones
2. Utiliza una estructura estándar de proyectos
3. Añade un fichero README al directorio raíz de tu proyecto
4. Utiliza un script maestro (‘makefile’)
5. Evitar guardar el espacio de trabajo (‘workspace’)
6. Aprovecha las ventajas de Rmarkdown
7. Aprovecha las herramientas que ayudan a escribir mejor código
8. Comenta tu código
9. Utiliza nombres memorables para los objetos
10. Documenta los datos
11. Comprueba los datos antes del análisis
12. Comprueba los resultados del análisis
13. Escribe código modular
14. Evita repeticiones en el código
15. Registra las dependencias