aeet
ASOCIACIÓN ESPAÑOLA DE ECOLOGÍA TERRESTRE

Ecosistemas 29(3):2129 [Septiembre-Diciembre 2020]
https://doi.org/10.7818/ECOS.2129

EcoNOTA

ecosistemas
REVISTA CIENTÍFICA DE ECOLOGÍA Y MEDIO AMBIENTE

ISSN 1697-2473

disponible en www.revistaecosistemas.net

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

F. Rodríguez-Sánchez[1,*]

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

* Autor de correspondencia: F. Rodríguez-Sánchez [f.rodriguez.sanc@gmail.com]

> Recibido el 23 de diciembre de 2020 - Aceptado el 23 de diciembre de 2020

Rodríguez-Sánchez, F. 2020. Quince consejos para mejorar nuestro código y flujo de trabajo con R. *Ecosistemas* 29(3): 2129. https://doi.org/10.7818/ECOS.2129

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 (Wilson et al. 2014, 2017; Rodríguez-Sánchez et al. 2016). En consecuencia, nuestro código es a menudo ineficiente, desordenado, propenso a errores, difícil de revisar y reutilizar.

En esta nota se recogen 15 recomendaciones para mejorar nuestro flujo de trabajo y programación particularmente en lenguaje R (R Core Team 2020) (aunque muchas de estas buenas prácticas sean igualmente aplicables a otros lenguajes). R se ha convertido en la herramienta estadística y lenguaje de programación más popular en ecología (Lai et al. 2019). Estas recomendaciones 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` (https://git-scm.com) 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 con plataformas como GitHub, GitLab o Bitbucket, se facilita enormemente el desarrollo colaborativo de proyectos (Blischak et al. 2016).

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 (**Fig.1**; Wilson et al. 2017; Cooper y Hsing 2017), por ejemplo aprovechando la infraestructura de proyectos de RStudio. Utiliza el paquete `here` (Müller 2020a) o `rprojroot` (Müller 2020b) para especificar las rutas a los diferentes archivos dentro del proyecto.

3. Añade un fichero README al directorio raíz de tu proyecto

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 (‘*makefile*’)

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")
```

Paquetes 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 espacio de trabajo (‘*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 (incluyendo tablas, figuras, etc) que reproducen todo el proceso de análisis. 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), 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 para los objetos

Utiliza nombres con significado que resuman el contenido o función del objeto (p. ej. `modelo_aditivo`, `modelo_interactivo` en lugar de `m1`, `m2`); ver https://style.tidyverse.org/syntax.html.

10. Documenta los datos

Prepara metadatos explicando qué representa cada variable (tipo de medida, unidades), autores, licencia de uso… Herramientas como `dataspice` (Boettiger et al. 2020) facilitan enormemente esta tarea, y mejoran la visibilidad y potencial de reutilización de los datos.

11. Comprueba los datos antes del análisis

En cualquier proyecto puede ocurrir que los datos de partida contengan errores (introducidos al teclear los datos, importarlos o manipularlos). Paquetes como `assertr` (Fischetti 2020), `validate` (van der Loo y de Jonge 2019) o `pointblank` (Iannone y Vargas 2020) resultan muy útiles para comprobar la calidad de los datos antes del análisis.

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.

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, o escribiendo funciones con un cometido específico e independientes del código principal del análisis.

14. Evita repeticiones en el código

A menudo necesitamos ejecutar unas líneas de código repetidamente. Por ejemplo, para producir 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))
```

¿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 para documentar las dependencias de nuestro análisis, desde la función `sessionInfo`, paquetes como `automagic` (Brokamp 2019) o `renv` (Ushey 2020) que registran 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.

```
- README                # información general del proyecto
- LICENSE               # licencia de uso
- deps.yaml             # registro de dependencias
- makefile              # script maestro que ejecuta los análisis
- data/
  |- raw_data/          # datos brutos
  |- clean_data/        # datos depurados
- analysis/
  |- reports/           # Documentos Rmarkdown
      |- data_prep.Rmd
      |- modelling.Rmd
      |- report.Rmd
      |- references.bib # bibliografía
  |- figures/           # Código y figuras finales
```

Figura 1. *Ejemplo de estructura de proyecto. Existe un fichero README (normalmente en formato markdown) con información general del proyecto (**recomendación nº 3**), un fichero especificando la licencia de uso de los datos y/o código, un fichero registrando las dependencias (**recomendación nº 15**), un script maestro o 'makefile' que ejecuta los distintos pasos del análisis en el orden correcto (**recomendación nº 4**), una carpeta de datos separando datos brutos y procesados, y una carpeta de análisis que contiene el código para generar las figuras finales y documentos Rmarkdown (**recomendación nº 6**) con los distintos pasos del análisis (**recomendación nº 13**).*

Agradecimientos

Al Integrative Ecology Group, por incitar a la escritura de esta nota, y al grupo de Ecoinformática de la AEET (en particular a Antonio Pérez-Luque, Ruth Delgado, Hugo Saiz, Alfonso Garmendia, Aitor Ameztegui, David García-Callejas e Ignasi Bartomeus), por sus sugerencias para mejorarla.

Referencias

Bertin, A.M., Baumer, B.S. 2020. Creating optimal conditions for reproducible data analysis in R with 'fertile'. *Stat*. https://doi.org/10.1002/sta4.332

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

Boettiger, C., Chamberlain, S., Fournier, A., Hondula, K., Krystalli, A., Mecum, B., et al. 2020. *dataspice: Create Lightweight Schema.org Descriptions of Data*. https://CRAN.R-project.org/package=dataspice

Brokamp, C. 2019. *automagic: Automagically Document and Install Packages Necessary to Run R Code*. https://CRAN.R-project.org/package=automagic

Bryan, J., Hester, J. 2019. *What They Forgot to Teach You About R*. https://rstats.wtf

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

Fischetti, T. 2020. *assertr: Assertive Programming for R Analysis Pipelines*. https://CRAN.R-project.org/package=assertr

Henry, L., Wickham, H. 2020. *purrr: Functional Programming Tools*. https://CRAN.R-project.org/package=purrr

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

Iannone, R., Vargas, M. 2020. *pointblank: Validation of Local and Remote Data Tables*. https://CRAN.R-project.org/package=pointblank

Lai, J., Lortie, C.J., Muenchen, R.A., Yang, J., Ma, K. 2019. Evaluating the popularity of R in ecology. *Ecosphere* 10: e02567.

Landau, W.M. 2020. *targets: Dynamic Function-Oriented 'Make'-Like Declarative Workflows*. https://wlandau.github.io/targets/

Landau, W.M. 2018. The drake R package: a pipeline toolkit for reproducibility and high-performance computing. *Journal of Open Source Software* 3(21):550.

Lau, M. 2020. *Rclean: A Tool for Writing Cleaner, More Transparent Code*. https://github.com/MKLau/Rclean

Müller, K. 2020a. *here: A Simpler Way to Find Your Files*. https://CRAN.R-project.org/package=here

Müller, K. 2020b. *rprojroot: Finding Files in Project Subdirectories*. https://CRAN.R-project.org/package=rprojroot

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

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

Rodríguez-Sánchez, F., Pérez-Luque, A.J., Bartomeus, I., Varela, S. 2016. Ciencia reproducible: qué, por qué, cómo? *Ecosistemas* 25: 83-92.

Ushey, K. 2020. *renv: Project Environments*. https://CRAN.R-project.org/package=renv

van der Loo, M., de Jonge, E. 2019 (en prensa). Data Validation Infrastructure for R. *Journal of Statistical Software*.

Wilson, G., Aruliah, D.A., Brown, C.T., Hong, N.P.C., Davis, M., Guy, R.T., et al. 2014. Best Practices for Scientific Computing. *PLoS Biology* 12: e1001745.

Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., Teal, T.K. 2017. Good enough practices in scientific computing. *PLOS Computational Biology* 13: e1005510.

ECOSISTEMAS es una revista editada por: aeet ASOCIACIÓN ESPAÑOLA DE ECOLOGIA TERRESTRE con la colaboración de: GOBIERNO DE ESPAÑA VICEPRESIDENCIA CUARTA DEL GOBIERNO MINISTERIO PARA LA TRANSICIÓN ECOLÓGICA Y EL RETO DEMOGRÁFICO Fundación Biodiversidad