Notas de Programación en R

Rubén Fernández Casal (rubenfcasal@gmail.com)

Edición: Marzo de 2023. Impresión: 2023-03-28

Índice general

Prólogo

Este es un libro con notas personales sobre programación en R para el análisis de datos, en el que incluyen referencias a información y recursos adicionales (se asumen unos conocimientos básicos de R). El contenido está sesgado por la experiencia personal (es mi forma de programar en R) pero puede resultar útil para otras personas. Cualquier sugerencia de mejora será bien recibida.

Este libro ha sido escrito en R-Markdown empleando el paquete `bookdown` y está disponible en el repositorio Github: rubenfcasal/notasr. Se puede acceder a la versión en línea a través del siguiente enlace:

https://rubenfcasal.github.io/notasr.

donde puede descargarse en formato pdf.

Para seguir los ejemplos mostrados en el libro se recomienda tener instalados los siguientes paquetes (realmente no se emplean todos): `Rcmdr`, `caret`, `tidymodels`, `tidyverse`, `openxlsx`, `DT`, `rmarkdown`, `knitr`, `remotes`, `devtools`. Por ejemplo mediante los siguientes comandos:

```
pkgs <- c("Rcmdr", "caret", "tidymodels", "tidyverse", "openxlsx", "DT",
          "rmarkdown", "knitr", "remotes", "devtools")
install.packages(setdiff(pkgs, installed.packages()[,"Package"]), dependencies = TRUE)
```

(puede que haya que seleccionar el repositorio de descarga, e.g. *Oficina de software libre (CIXUG)*).

El código anterior no reinstala los paquetes ya instalados, por lo que podrían aparecer problemas debidos a incompatibilidades entre versiones (aunque no suele ocurrir, salvo que nuestra instalación de R esté muy desactualizada). Si es el caso, en lugar de la última línea se puede ejecutar:

```
install.packages(pkgs, dependencies = TRUE) # Instala todos...
```

Para generar el libro (compilar) serán necesarios paquetes adicionales, para lo que se recomendaría consultar el libro de "Escritura de libros con bookdown" en castellano.

Capítulo 1

Introducción

Como aparece en el prólogo, estos apuntes contienen recomendaciones y notas personales sobre programación en R para el análisis de datos, en el que incluyen referencias a información y recursos adicionales que considero de interés. Se tratará de mostrar una forma de llevar a cabo las distintas tareas que pueden surgir en el análisis de datos empleando R, esto no quiere decir que sea la mejor forma de hacerlo o la más cómoda (que dependerá de cada persona).

En estas notas ***se asumen unos conocimientos básicos de R***, un lenguaje de programación (interpretado) y un entorno estadístico desarrollado específicamente para el análisis estadístico. Puede ser una herramienta de gran utilidad a lo largo de todo el proceso de obtención de información a partir de datos (ver Figura 1.1).

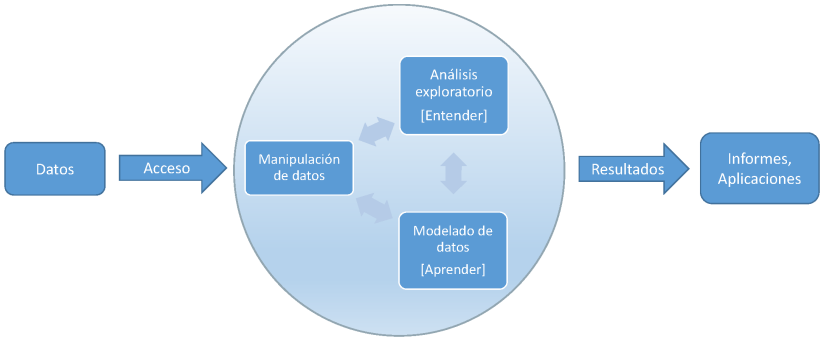

Figura 1.1: Etapas del proceso

Para una introducción a la programación en R se puede consultar el libro:

Fernández-Casal R., Roca-Pardiñas J., Costa J. y Oviedo-de la Fuente M. (2022). *Introducción al Análisis de Datos con R* (github).

Adicionalmente, en este post se incluyen enlaces a recursos adicionales, incluyendo libros y cursos, que pueden ser útiles para el aprendizaje de R.

El primer paso es la instalación de R, para ello se recomienda seguir los pasos en este post.

Para el desarrollo de código e informes la recomendación es emplear *RStudio Desktop*, que se puede instalar y configurar siguiendo las indicaciones en este post. También puede resultar de interés consultar:

- RStudio cheatsheet
- Using the RStudio IDE

Sin embargo, en ciertos casos puede ser recomendable ejecutar el código R directamente desde una ventana de comandos (por ejemplo para ejecutar varios programas de forma simultánea en distintos

directorios de trabajo o si los requerimientos computacionales son grandes). En mi caso, cuando trabajo en Windows, acostumbro a emplear el explorador para situarme en el directorio donde quiero ejecutar código y abrir una ventana de comandos, escribiendo `cmd` en el cuadro superior donde se muestra la ruta. Posteriormente, como añadí en directorio de instalación de R al *path* (ver post), ejecuto[1] `R` y finalmente un comando de la forma:

```
source("mi_script.R", echo = TRUE, encoding = "UTF-8") # UTF-8 importante en R < 4.2
```

1.1 Organización

Para la organización de archivos (datos, código, informes…) lo recomendable es emplear un directorio con la estructura adecuada.

Dependiendo del objetivo puede interesar emplear un proyecto de RStudio (menú *File > New project…*). En mi caso empleo esta opción para paquetes, libros en bookdown, webs con blogdown y aplicaciones shiny. En otros casos empleo una carpeta que puede tener subdirectorios (si el proyecto es más grande) para distintos tipos de archivos o para distintas tareas (con el objetivo de facilitar la búsqueda). Por ejemplo: *datos*, *informes*, *resultados_2023*…

Mi recomendación es emplear nombres de archivos y carpetas en minúscula (o con la primera letra en mayúsculas) y sin espacios (por ejemplo empleando `_` para separar palabras o iniciales). Los nombres deberían ser lo más descriptivos posibles (en el sentido de evitar confusión). Pueden incluirse descripciones más completas en el código, en ficheros de texto (e.g. *Descripción_archivos.txt*), o incluso en hojas de cálculo. Yo además acostumbro a incluir archivos del tipo *Notas.txt* (con recordatorios, decisiones…) o *Pendente.txt* (con próximos pasos, mejoras o verificaciones pendientes…).

Además, nos puede interesar establecer opciones de R específicas para el proyecto (por ejemplo opciones de configuración de memoria, de paquetes o variables de entorno, incluyendo claves privadas), de forma que se establezcan automáticamente al iniciar R o RStudio. Para más detalles ver la ayuda de `?Startup` Managing R with .Rprofile, .Renviron, Rprofile.site, Renviron.site, rsession.conf, and repos.conf

1.1.1 Código e informes

Mi recomendación a la hora de escribir código es seguir un **proceso iterativo**. Se comienza realizando pruebas y al finalizar cada etapa se trata de reorganizar el código (adaptándolo al estilo de programación elegido, lo que incluiría añadir comentarios y secciones) de forma que sea más cómodo continuar trabajando en siguientes etapas (y si es posible que resulte más fácil de adaptar para otros casos).

En el caso de informes el proceso sería similar, empleando como punto de partida un fichero de código en formato spin (ver e.g. Apéndice), en el que el texto RMarkdown se incluye como un comentario de código empleando `#'`. Por ejemplo:

```
#' # Sección
#'
#' ## Subsección
#'
#' Texto rmarkdown...
```

En primer lugar me preocupo de escribir un código funcional y, además de ir añadiendo comentarios de la forma habitual, voy añadiendo secciones y texto rmarkdown en formato spin. Finalmente, cuando tengo una primera versión del código (que puedo ir previsualizando; en RStudio basta con pulsar *Ctrl + Shift + K*, o el botón correspondiente en la barra superior, o seleccionar *File > Knit Compile Report…*), lo transformo a formato *.Rmd* con un comando de la forma:

[1] También se puede ejecutar un script de R de forma no interactiva ejecutando en el intérprete de comandos del sistema operativo: `R CMD BATCH [opciones] mi_script.R [fichero_salida]` (cambiando `R` por la ruta completa, e.g. `"C:\Program Files\R\R-4.2.1\bin\R.exe"`, si no se añadió al path. También se puede incluir en un fichero *.bat*, para poder ejecutarlo repetidas veces con mayor facilidad). Ver Appendix B Invoking R de Introduction to R para información sobre las distintas opciones.

```
knitr::spin("Informe.R",knit = FALSE)
```

donde termino de redactar (`knitr::purl("Informe.Rmd", documentation = 2)` genera un nuevo fichero *Informe.R* donde resulta más cómodo modificar o desarrollar código).

Se recomienda **elegir un estilo que sea consistente y seguirlo por completo** en todo el proyecto. Lo principal sería el operador de asignación y el estilo de nombres (de objetos, variables o ficheros):

- `estilo.clasico`: es el estilo del paquete base de R. Muchos programadores no lo recomiendan (principalmente porque este separador no se admite en otros lenguajes y porque puede dar lugar a confusión con métodos S3, ver Sección 2.3).
- `estilo_serpiente` (o `Estilo_serpiente`): es el estilo de la colección de paquetes `tidyverse`.
- `EstiloCamello` (o `estiloCamello`): es el estilo (casi obligatorio) para las clases `R6` (ver Sección 2.3). El paquete `shiny` emplea la variante que comienza por minúsculas.

Recomiendo emplear `<-` como operador de asignación y escribir todos los **nombres en minúsculas**. Yo tengo tendencia a emplear el `estilo.clasico`, sobre todo si el código no depende de paquetes tidyverse (en ese caso suelo emplear `estilo_serpiente`). También influye el estilo de nombres empleado por la fuente de datos o el requerido en los resultados.

El estilo también debe especificar el sangrado, el espaciado, etc. Por ejemplo:

- Tidyverse style guide
- Google's R Style Guide

Además se recomienda **crear secciones y documentar el código adecuadamente**. En RStudio se puede crear una sección pulsando *Ctrl + Shift + R* o añadiendo al menos 4 guiones (`-`, también `=` o `#`) después de un comentario. Por ejemplo:

```
# Sección ----
## Subsección ----
```

El orden de las secciones y subsecciones es importante. Al principio del código debería ir:

1. Los parámetros o variables globales.
2. La carga de paquetes (únicamente los mínimos requeridos).
3. La carga de código externo.
4. La carga de archivos de datos (o al principio de la sección donde se emplean, si son datos auxiliares).

No se recomienda emplear rutas absolutas en el código, del tipo:

```
setwd("C:/Documentos/Proyectos/Proyecto_X")
load("C:/Documentos/Proyectos/Proyecto_X/datos_x.RData")
source("C:/Documentos/Proyectos/R/Herramientas.R")
```

Como punto de partida el directorio de trabajo debería ser la carpeta del proyecto. Esto ya ocurre por defecto si empleamos proyectos de RStudio o si iniciamos RStudio abriendo un archivo de código en esta carpeta. En general, la recomendación es asumir que el directorio de trabajo es aquel en el que se encuentra el archivo de código (lo que también ocurre por defecto al compilar un documento RMarkdown). Si no es el caso se puede emplear el menú *Sesion > Set Working Directory > To Source File Location*.

Para establecer la ruta a archivos o directorios **se recomienda emplear rutas relativas** (usando *../* para acceder a la carpeta anterior; *./* sería el directorio actual de trabajo). Por ejemplo:

```
load("datos/datos_x.RData")
source("../R/Herramientas.R")
fecha_txt <- as.character(Sys.Date() - 1, format = "%m_%d") # Por ejemplo...
rmarkdown::render("informe.Rmd",
                  output_file = paste0('informes/informe_', fecha_txt, '.html'),
```

```
                  # params = list(fecha_txt = fecha_txt),
                  envir = new.env(), encoding = "UTF-8")
```

La mejor forma de organizar funciones es desarrollar un paquete, como se comenta más adelante en la Sección 2.4.

Para desarrollar código de forma colaborativa, la recomendación es emplear un sistema de control de versiones. Se puede configurar RStudio para emplear Git (ver Happy Git and GitHub for the useR y Git and GitHub), sin embargo yo prefiero emplear GitHub Desktop.

1.1.2 Datos

La recomendación es emplear ficheros de datos con el formato por defecto de R (datos binarios comprimidos), con extensión *.RData*. Hay que tener en cuenta que lo esperable es que el archivo contenga un conjunto de datos con el mismo nombre, aunque podría no ser el caso e incluso contener varios objetos.

Uno de los problemas con los ficheros *.RData* es que, al cargarlos con `load()` de la forma habitual, se añaden al entorno de trabajo los objetos que contienen con los nombres con que se almacenaron (y si ya existe alguno con ese nombre lo sobreescribe) Para almacenar un único objeto de forma que se pueda cargar posteriormente especificando el nombre, se pueden emplear las funciones `saveRDS()` y `readRDS()`.

Sin embargo, lo habitual es que inicialmente los datos procedan de una fuente externa. Se pueden importar datos externos en casi cualquier formato a R (aunque puede requerir instalar paquetes adicionales). Mi recomendación es separar los análisis de la importación de los datos. Crear un fichero de código específicamente para importar los datos[2], hacer el (pre)procesado y guardarlos en formato *.RData*. Yo habitualmente empleo el mismo nombre para el archivo de código y el archivo de datos que se genera (e.g. *datos.R* contiene el código necesario para generar *datos.RData*; no suelo renombrar el fichero fuente de datos externo, aunque se aleje mucho del estilo elegido). Asociado a un mismo conjunto de datos puede haber distintos archivos de código para realizar distintos análisis (el nombre de esos archivos debería dar una pista del análisis que realizan).

En muchas ocasiones, para modificar los nombres de las variables o los niveles de un factor, suelo recurrir a la función `dput()` para escribirlos en modo texto (e.g. `dput(tolower(names(data)))`) y posteriormente modificarlos a mano.

Yo recomiendo añadir un atributo `variable.labels` que contenga un vector de etiquetas de las variables y empleando como nombres de las componentes las propias variables:

```
data(cars)
# dput(names(cars))
variable.labels <- c(speed = "Speed (mph)", dist = "Stopping distance (ft)")
attr(cars, "variable.labels") <- variable.labels
str(cars)
```

```
## 'data.frame':    50 obs. of  2 variables:
##  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
##  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
##  - attr(*, "variable.labels")= Named chr [1:2] "Speed (mph)" "Stopping distance (ft)"
##   ..- attr(*, "names")= chr [1:2] "speed" "dist"
```

```
# View(cars)
# with(cars, plot(speed, dist, xlab = variable.labels["speed"],
#                 ylab = variable.labels["dist"]))
```

Para leer ficheros de Excel acostumbro a utilizar los paquetes `openxlsx` (solo para archivos con extensión *.xlsx*) o `readxl` (colección `tidyverse`). En estos casos además se puede añadir una nueva

[2]Con algunos tipos de datos, se puede emplear los submenús de RStudio *File > Import Dataset* para seleccionar los ajustes, previsualizando el resultado, y generar el código para importarlos.

hoja de cálculo con los nombres de las variables junto con su etiqueta, que se puede cargar y emplear durante el preprocesado. Adicionalmente esta tabla puede incluir una columna con los nuevos nombres (yo recomiendo no modificar los antiguos en este fichero), otra con un filtro para seleccionar variables (o el orden después del procesado) e incluso una columna con anotaciones o observaciones.

Parte I

R

Capítulo 2

El lenguaje R

Cualquier análisis de R requiere programación, aunque normalmente se puede llevar a cabo sin conocimientos profundos del lenguaje (*useR*). Sin embargo, para desarrollar nuevas herramientas de forma efectiva (*programeR*) es necesario tener una idea del funcionamiento interno de R. La referencia recomendada para usuarios de R que deseen mejorar sus conocimientos de programación y comprensión del lenguaje es:

Wickham, Hadley (2019). *Advanced R, 2ª edición*, Chapman & Hall, 1ª edición.

También puede ser de utilidad el manual R Language Definition para consultas adicionales[1].

2.1 Paquetes

Al instalar R se instalan los denominados **paquetes base** y (por defecto) los **paquetes recomendados** por los desarrolladores de R (el *R Core Team*). Podemos acceder a la lista de paquetes instalados:

```
pkgs <- installed.packages()
names(which(pkgs[ ,"Priority"] == "base"))
```

```
##  [1] "base"      "compiler"  "datasets"  "graphics"  "grDevices" "grid"
##  [7] "methods"   "parallel"  "splines"   "stats"     "stats4"    "tcltk"
## [13] "tools"     "utils"
```

```
names(which(pkgs[ ,"Priority"] == "recommended"))
```

```
##  [1] "boot"       "class"      "cluster"    "codetools"  "foreign"
##  [6] "KernSmooth" "lattice"    "MASS"       "Matrix"     "mgcv"
## [11] "nlme"       "nnet"       "rpart"      "spatial"    "survival"
```

Para instalar paquetes adicionales se puede emplear `install.packages()`. Por ejemplo:

```
pkgs <- c("Rcmdr", "caret", "tidymodels", "tidyverse", "remotes", "devtools",
          "sf", "gstat", "geoR", "quadprog", "DEoptim", "spam", "openxlsx",
            "bookdown", "blogdown", "pkgdown")
install.packages(setdiff(pkgs, installed.packages()[,"Package"]), dependencies = TRUE)
```

En Windows (y en MacOS) esta función instala por defecto paquetes compilados (`type = "binary"`, que dependen del sistema operativo y de la versión R) disponibles en CRAN. Aunque podría instalar paquetes disponibles en otros repositorios. Por ejemplo:

```
url <- "https://github.com/rubenfcasal/simres/releases/download/v0.1/simres_0.1.3.zip"
install.packages(url, repos = NULL)
```

[1] Los manuales oficiales también están disponibles en formato bookdown en este post.

También se pueden instalar paquetes directamente a partir del código fuente con `type = "source"` (por defecto en Linux), pero en ciertos casos es necesario tener instaladas herramientas adicionales (por ejemplo Rtools en Windows si el paquete contiene código en C, C++ o Fortran). Esto permitiría incluso instalar paquetes retirados de CRAN (e.g. actualmente `kedd`), ya que siempre se mantiene el código (en un archivo comprimido de la forma `paquete_x.y.z.tar.gz`).

Si se quieren instalar paquetes de repositorios distintos de CRAN (GitHub, GitLab, Bitbucket, …), puede ser recomendable instalar `remotes`. Por ejemplo:

```
remotes::install_github("rubenfcasal/simres", INSTALL_opts = "--with-keep.source")
```

Además puede ser de utilidad mantener los comentarios originales del paquete para entender mejor el código (por ejemplo si se quiere modificar).

Otras funciones que pueden ser de interés son: `remove.packages()`, `update.packages()` y `available.packages()`.

Al iniciar el programa `R` se cargan por defecto en memoria los principales paquetes base, añadiéndolos a la ruta de búsqueda (a continuación del entorno de trabajo `.GlobalEnv` y siempre terminando con en el paquete `base`, el primero que se carga):

```
search()
```

```
## [1] ".GlobalEnv"        "package:stats"     "package:graphics"
## [4] "package:grDevices" "package:utils"     "package:datasets"
## [7] "package:methods"   "Autoloads"         "package:base"
```

Concretamente se añade a la ruta de búsqueda un entorno que contiene el conjunto de objetos exportables del paquete (definido en el denominado *namespace* del paquete). Esta ruta determina los objetos visibles en el entorno global y el orden en se buscan (para más detalles ver 7.2 Environment basics y 7.4 Special environments de Advanced R).

Podemos cargar paquetes adicionales (previamente instalados) con `library()` o `require()`, por ejemplo:

```
if (!require(knitr)) {
  install.packages("knitr")
  library(knitr)
}
spin("01-Introduccion.R", knit = FALSE)
```

Aunque **no se recomienda que el código instale automáticamente paquetes** (en general que haga cambios en la configuración del equipo en el que se ejecuta).

Al cargar un paquete se añade por defecto en la segunda posición de la ruta de búsqueda (justo después del entorno global, desplazando al resto). También se podrían añadir otros objetos, por ejemplo data.frames, con la función `attach()` pero **no se recomienda** (se puede utilizar `with()` como alternativa).

Hay que tener cuidado con las versiones instaladas de los paquetes:

```
packageVersion("dplyr")
```

```
## [1] '1.0.10'
```

y con sus dependencias (los paquetes tienen su propia ruta de búsqueda, determinada por el *namespace* del paquete). Al actualizar o instalar nuevos paquetes pueden aparecer problemas al ejecutar código antiguo (a veces al trabajar en nuevos proyectos acabamos haciendo que los antiguos dejen de funcionar).

Se puede instalar versiones específicas de un paquete con `install_version()`:

```
remotes::install_version("dplyr", version = "1.11") # repos = "https://ftp.cixug.es/CRAN")
```

Para asegurarse que el código de un proyecto se puede ejecutar a lo largo del tiempo se puede emplear el paquete `renv` (se puede configurar automáticamente al crear un proyecto de RStudio). Este paquete permite registrar las versiones exactas de los paquetes de los que depende un proyecto y volver a instalarlas (incluso en otro equipo) si es necesario. Para más detalles ver la viñeta Introduction to renv.

Sin embargo de esta forma aún dependemos del sistema operativo que deberíamos configurar adecuadamente. La recomendación para que un proyecto en R (por ejemplo una aplicación shiny) se pueda ejecutar en cualquier equipo, es emplear un contenedor docker. Para más detalles ver Docker overview y [The Rocker Project]https://rocker-project.org/).

2.2 Funciones

> "Everything that happens in R is the result of a function call".
>
> — John M. Chambers

Como es bien conocido, en R se pueden asignar los argumentos de una función por posición o por nombre (del correspondiente parámetro en la definición de la función, denominado *argumento formal en R*). En general, la recomendación es asignar los argumentos por nombre:

```
funcion(parametro1 = argumento1, parametro2 = argumento2, ...)
```

De esta forma no importa el orden de los parámetros y, por ejemplo, evitaremos problemas si en el futuro hay cambios en la definición de la función. Los parámetros pueden tener valores por defecto y solo sería necesario especificarlos para asignarles un valor distinto.

Podemos llamar a una función de un paquete sin necesidad de cargarlo (añadirlo a la ruta de búsqueda) empleando `paquete::funcion`. Esto es especialmente recomendable al desarrollar nuevas funciones (es un requisito para subir paquetes a CRAN), ya que de esta forma se evitan conflictos entre funciones con el mismo nombre en paquetes distintos. Por ejemplo:

```
if (!requireNamespace(knitr)) stop("'knitr' package required")
knitr::spin("01-Introduccion.R", knit = FALSE)
```

Hay que tener en cuenta que R emplea Lazy evaluation, los argumentos no se evalúan hasta que se necesitan (lo cual puede producir mensajes de error inesperados, pero también permite añadir funcionalidades adicionales empleando la denominada evaluación no estándar o metaprogramación).

R es un lenguaje interpretado y podemos evaluar expresiones empleando código. Por ejemplo, podemos reproducir el proceso de introducir un comando en la consola con las funciones `eval()` y `parse()`:

```
eval(parse(text = "1:10"))
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
distr <- "norm"  # "unif", "exp", "t"
ddistr <- eval(parse(text = paste0("d", distr)))
# str(ddistr)
# curve(ddistr(x, 0, 0.5), -3, 3)
```

Para llamar a una función especificando los parámetros de forma dinámica (empleando una lista) podemos emplear `do.call()`. Por ejemplo:

```
# Listar ficheros csv
files.csv <- dir(path = "datos", pattern = "*.csv", full.names = TRUE)
# Leer datos a una lista
# (suponemos variante local con ; para separar valores)
data.list <- lapply(files.csv, read.csv2)
# Combinar
datos <- do.call('rbind', data.list)
```

Hay que tener en cuenta que las funciones tienen su propio entorno y su propia ruta de búsqueda, determinada por el entorno donde se crearon (el *namespace* en el caso de las funciones de un paquete). Esto es lo que se conoce como Lexical scoping.

```
x <- 1
addx <- function(y) {
  x + y
}
addx(10)
```

```
## [1] 11
```

```
addx10 <- function() {
  x <- 10   # x <<- 10   # assign("x", 10, envir = .GlobalEnv)
  addx(x)
}
addx10()
```

```
## [1] 11
```

```
x
```

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

2.3 Programación orientada a objetos (funciones genéricas)

> "Everything that exists in R is an object".
>
> — John M. Chambers

R implementa programación orientada a objetos (OOP). Por ejemplo, es bien conocido que algunas funciones (entre ellas `print()`, `plot()` o `summary()`) se comportan de manera diferente dependiendo de la clase (el tipo de objeto) de sus argumentos, son las denominadas *funciones genéricas*.

Realmente R dispone de varios sistemas de OOP, entre ellos podríamos destacar (ver capítulos en Object-oriented programming de Advanced R):

- S3: Es un sistema muy simple, las clases no tienen una definición formal. Es el empleado en el paquete `base` de R y en la mayoría de paquetes que usan OOP. Descrito inicialmente en:

  Becker R.A., Chambers J.M. y Wilks A.R. (1988), *The New S Language: A Programming Environment for Data Analysis and Graphics* (A.K.A. the *Blue Book*). Chapman & Hall.

  Chambers J.M. y Hastie T.J. eds. (1992), *Statistical Models in S* (A.K.A. the *White Book*). Chapman & Hall.

- S4 (**no lo recomiendo**): Está implementado en el paquete `methods` (uno de los *paquetes base*) de R. Descrito inicialmente en:

  Chambers J.M. (1998), *Programming with Data* (A.K.A. the *Green Book*). Springer.

- R6: Es un sistema OOP encapsulado similar al de otros lenguajes de programación. Está implementado en el paquete `R6` (no se instala por defecto).

Yo en principio **recomendaría usar el sistema S3**, aunque es bastante rudimentario y puede resultar inicialmente confuso a programadores con experiencia en otros lenguajes. En cualquier caso es muy recomendable conocer su funcionamiento. Este sistema esta basado en funciones genéricas. La clase es un atributo de los objetos (*encapsulación*), una cadena de texto o un vector de cadenas (*herencia*), al que se puede acceder con la función `class()`. A partir de la clase del argumento, la función genérica determina el método (función especializada) al que debe llamar (*polimorfismo*). En S3 el despacho de métodos (*method dispatch*) es muy simple, si la función genérica es `generica()` y la clase del primer argumento es `"clase"`, se llama a la función (método) `generica.clase()` si existe. Si la clase del objeto es heredada (un vector de cadenas), se van buscando los métodos por orden de parentesco y si no se encuentra ninguno, se llama al método por defecto `generica.default()` (se llama a la

primera función de `paste0("generica.", c(class(x), "default"))` que se encuentre en la ruta de búsqueda; podríamos reemplazarla…).

La función genérica suele ser muy sencilla, básicamente incluye una llamada a `UseMethod("generica")`. Por ejemplo:

```
plot
```

```
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x0000020c8dacf898>
## <environment: namespace:base>
```

Podemos obtener los métodos asociados a una función genérica con `methods(genérica)`. Por ejemplo:

```
methods(plot)
```

```
##  [1] plot.acf*           plot.data.frame*    plot.decomposed.ts*
##  [4] plot.default        plot.dendrogram*    plot.density*
##  [7] plot.ecdf           plot.factor*        plot.formula*
## [10] plot.function       plot.hclust*        plot.histogram*
## [13] plot.HoltWinters*   plot.isoreg*        plot.lm*
## [16] plot.medpolish*     plot.mlm*           plot.ppr*
## [19] plot.prcomp*        plot.princomp*      plot.profile.nls*
## [22] plot.raster*        plot.spec*          plot.stepfun
## [25] plot.stl*           plot.table*         plot.ts
## [28] plot.tskernel*      plot.TukeyHSD*
## see '?methods' for accessing help and source code
```

Podemos acceder a la ayuda del correspondiente método de la forma habitual (e.g. `?plot.lm`), pero puede que algunos métodos no sean objetos definidos como exportables en el *namespace* del paquete que los implementa (los marcados con un `*`) y por tanto no son en principio accesibles para el usuario. Siempre podemos acceder a ellos empleando `paquete:::metodo` o `getAnywhere(metodo)` (e.g. `stats:::plot.lm` o `getAnywhere(plot.lm)`).

Para listar los métodos disponibles para una clase, podemos emplear el parámetro `class`. Por ejemplo:

```
methods(class = "lm")
```

```
##  [1] add1           alias          anova          case.names     coerce
##  [6] confint        cooks.distance deviance       dfbeta         dfbetas
## [11] drop1          dummy.coef     effects        extractAIC     family
## [16] formula        hatvalues      influence      initialize     kappa
## [21] labels         logLik         model.frame    model.matrix   nobs
## [26] plot           predict        print          proj           qr
## [31] residuals      rstandard      rstudent       show           simulate
## [36] slotsFromS3    summary        variable.names vcov
## see '?methods' for accessing help and source code
```

Para una programación orientada a objetos más formal la recomendación es emplear el sistema R6.

2.4 Desarrollo de funciones y paquetes

La recomendación es documentar todas las funciones que se crean y preferiblemente empleando el formato `roxygen2`. Por ejemplo:

```
# read_excel_list(path, pattern, ...)
# ----------------------------------------
#' Lee los ficheros xls y xlsx de un directorio
#'
#' @param path Ruta al directorio con los ficheros excel
```

```
#' (por defecto el directorio de trabajo).
#' @param pattern Expresión regular empleada en la selección de ficheros
#' (ver `list.files()`).
#' @param ... Parámetros adicionales de `readxl::read_excel()`.
#' @return Una lista cuyas componentes son las correspondientes tablas de datos
#' (`tibble`) y con nombres los nombres de los archivos sin extensión.
#' @examples \dontrun{
#' data_list <- read_excel_list("datos") # "./datos"
#' data_all <- dplyr::bind_rows(data_list)
#' }
# Pendiente:
#   - Controlar posible error al leer
# ........................................
read_excel_list <- function(path = ".", pattern = "\\.(xls|xlsx)$", ...) {
  if (!requireNamespace(readxl))  stop("'readxl' package required")
  files <- dir(path, pattern = pattern, full.names = TRUE) # ?list.files
  data_list <- vector(length(files), mode = 'list')
  for (i in seq_along(files))
      data_list[[i]] <- readxl::read_excel(files[i], ...)
  data_names <- sub('\\.xlsx$', '', basename(files))
  names(data_list) <- data_names
  data_list
}
```

En muchas ocasiones se emplea como punto de partida una función implementada en R. En RStudio la forma más sencilla de obtener el código de la función es emplear `View(funcion)` (si la función es visible, en caso contrario `View(paquete:::funcion)`). Si la función llama a funciones internas (que no se exportan en el namespace) del paquete que la implementa, podríamos emplear también los tres dobles puntos, pero la recomendación sería descargar el código del paquete (si está en CRAN, un fichero comprimido de la forma `paquete_x.y.z.tar.gz` que se puede descargar en la sección *Downloads* de la web del paquete *https://CRAN.R-project.org/package=paquete*).

La mejor forma de organizar funciones es crear un paquete. Para ello se recomienda seguir:

Wickham, Hadley (2015). *R packages: organize, test, document, and share your code* (actualmente 2ª edición en desarrollo con H. Bryan), O'Reilly, 1ª edición.

También puede ser de utilidad el manual Writing R Extensions para información adicional.

Parte II

Tidyverse

Capítulo 3

La colección de paquetes tidyverse

En preparación...

En los capítulos de esta parte se pretende realizar una breve introducción al *ecosistema* **Tidyverse**, una colección de paquetes diseñados de forma uniforme (con la misma filosofía y estilo) para trabajar conjuntamente.

La referencia recomendada para usuarios de R que deseen iniciarse en el uso de estos paquetes es:

Wickham, H., y Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data*, online-castellano, O'Reilly.

El paquete `tidyverse` está diseñado para facilitar la instalación y carga de los paquetes principales de la colección tidyverse con un solo comando. Al instalar este paquete se instalan paquetes que forman el denominado núcleo de tidyverse (se cargan con `library(tidyverse)`):

- ggplot2: visualización de datos.
- dplyr: manipulación de datos.
- tidyr: reorganización (limpieza) de datos.
- readr: importación de datos.
- tibble: tablas de datos (modificación de `data.frame`).
- purrr: programación funcional.
- stringr: manipulación de cadenas de texto.
- forcats: manipulación de factores.
- lubridate: manipulación de fechas y horas.

y un conjunto de paquetes recomendados:

Referencias

Fernández-Casal R., Costa J. y Oviedo de la Fuente, M. (2021). *Aprendizaje Estadístico*. github.

Fernández-Casal R., Roca-Pardiñas J., Costa J. y Oviedo-de la Fuente M. (2023). *Introducción al Análisis de Datos con R*. ISBN: 978-84-09-41823-7. github.

Grolemund, G. (2014). *Hands-on programming with R: Write your own functions and simulations*, O'Reilly.

Kuhn, M. y Silge, J. (2022). *Tidy Modeling with R*. O'Reill.

Matloff, N. (2011). *The art of R programming: A tour of statistical software design*, No Starch Press.

Wickham, H. (2015). *R packages: organize, test, document, and share your code* (actualmente 2ª edición en desarrollo con H. Bryan), O'Reilly, 1ª edición.

Wickham, H. (2019). *Advanced R, 2ª edición*, Chapman & Hall, 1ª edición..

Wickham, H., y Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data*, online-castellano, O'Reilly.

NOTA: En la bibliografía complementaria se incluyen algunas de estas referencias, y una selección de libros en abierto, organizados por temas.

Bibliografía por temas

En preparación...

A continuación se muestra una selección de ***libros en abierto*** que considero que pueden resultar de utilidad. Para referencias adicionales recomiendo consultar:

- Baruffa, O. (2022). *Big Book of R*: Your last-ever bookmark (hopefully…).

Iniciación a la programación en R

- Wickham, H., y Grolemund, G. (2016). *R for data science: import, tidy, transform, visualize, and model data*, online-castellano, O'Reilly.
- Grolemund, G. (2014). *Hands-on programming with R: Write your own functions and simulations*, O'Reilly.
- Fernández-Casal R., Roca-Pardiñas J., Costa J., y Oviedo de la Fuente, M. (2022). *Introducción al Análisis de Datos con R*. github.

Programación avanzada en R

- Wickham, H. (2019). *Advanced R, 2ª edición*, Chapman & Hall, 1ª edición..
- Wickham, H. (2015). *R packages: organize, test, document, and share your code* (actualmente 2ª edición en desarrollo con H. Bryan), O'Reilly, 1ª edición.

Regresión y aprendizaje estadístico

- Fernández-Casal R., Costa J. y Oviedo de la Fuente, M. (2021). *Aprendizaje Estadístico*. github.
- Kuhn, M., y Silge, J. (2022). *Tidy Modeling with R*, O'Reilly.
- Fernández-Casal R., Cao R. y Costa J. (2023). *Técnicas de Simulación y Remuestreo* (github). La anterior edición (Fernández-Casal R. y Cao R., 2022, *Simulación Estadística*) está disponible en la rama *primera_edicion*.

Datos temporales y espaciales

- Fernández-Casal R. y Cotos-Yáñez T.R. (2021). *Estadística Espacial con R*. github.
- Hyndman, R.J., y Athanasopoulos, G. (2021). *Forecasting: principles and practice*. OTexts.
- Lovelace, R., Nowosad, J., y Muenchow, J. (2019). *Geocomputation with R*. CRC.
- Moraga, P. (2019). *Geospatial health data: Modeling and visualization with R-INLA and shiny*, CRC.
- Pebesma, E., y Bivand, R. (2021). *Spatial Data Science*.

Rmarkdown

- Fernández-Casal, R. y Cotos-Yáñez, T.R. (2018). *Escritura de libros con bookdown*, github. Incluye un apéndice con una Introducción a RMarkdown.

Enlaces

Repositorio: rubenfcasal/notasr

Recursos para el aprendizaje de R: En este post se muestran algunos recursos que pueden ser útiles para el aprendizaje de R y la obtención de ayuda.

Bookdown:

Introducción a RMarkdown.

Posit (RStudio)

- Blog
- Videos
- Chuletas (Cheatsheets)
- ***tidyverse***:
  - dplyr
  - tibble
  - tidyr
  - stringr
  - readr
  - Best Practices in Working with Databases
- tidymodels
- sparklyr
- shiny