

# Aprendiendo R sin morir en el intento

Javier Álvarez Liébana

Última actualización: 08-09-2021



# Contents

<b>I Toma de contacto</b>	<b>19</b>
<b>1 Instalación</b>	<b>21</b>
1.1 Instalación de RStudio . . . . .	25
1.2 Instalación de paquetes . . . . .	29
1.3 Consejos y tips . . . . .	31
<b>2 Primeros pasos</b>	<b>35</b>
2.1 Excel no es tu amigo . . . . .	36
2.2 Primeros pasos: calculadora . . . . .	41
2.3 Tipos de errores . . . . .	44
2.4 Consejos y tips . . . . .	45
2.5 □ Ejercicios . . . . .	46
<b>II Conceptos básicos</b>	<b>49</b>
<b>3 Tipos de datos I: vectores</b>	<b>51</b>
3.1 Vectores numéricos . . . . .	51
3.1.1 Secuencias con un patrón . . . . .	52

3.2	Operaciones aritméticas con vectores numéricos . . . . .	55
3.3	Operaciones estadísticas con vectores numéricos . . . . .	56
3.4	Vectores de caracteres (texto) . . . . .	57
3.5	Vectores lógicos (TRUE/FALSE) . . . . .	61
3.6	Datos ausentes: NA y NaN . . . . .	64
3.7	Seleccionar elementos de un vector . . . . .	67
3.7.1	which . . . . .	72
3.7.2	NULL . . . . .	73
3.8	Ordenar vectores . . . . .	74
3.9	Fechas . . . . .	75
3.10	□ Glosario . . . . .	76
3.11	Consejos y tips . . . . .	77
3.12	□ Ejercicios . . . . .	82
<b>4</b>	<b>Flujo de trabajo: proyecto</b>	<b>87</b>
4.1	Crear proyecto . . . . .	87
4.2	Directorios de trabajo y cabecera . . . . .	92
4.3	Ejecución . . . . .	97
4.4	Consejos y tips . . . . .	98
<b>5</b>	<b>Tipos de datos II: tablas</b>	<b>101</b>
5.1	Matrices . . . . .	101
5.2	Apply vs bucles . . . . .	106
5.3	Tablas: data.frames . . . . .	109
5.3.1	Data.frames: selección manual de columnas y filas . . . . .	114
5.4	Consejos y tips . . . . .	121

CONTENTS	5
5.5 □ Ejercicios . . . . .	123
<b>6 Importando/exportando</b>	<b>133</b>
6.1 Importación de datos . . . . .	133
6.1.1 Archivo .RData . . . . .	134
6.1.2 Archivo .csv . . . . .	135
6.1.3 Archivo .xlsx . . . . .	140
6.1.4 Desde web . . . . .	141
6.2 Exportación de datos . . . . .	143
6.2.1 Guardar en .RData . . . . .	143
6.2.2 Guardar en .csv . . . . .	144
6.3 Consejos y tips . . . . .	144
<b>7 Estructuras de control</b>	<b>149</b>
7.1 if...else . . . . .	149
7.2 for/while . . . . .	152
7.2.1 BREAK/NEXT . . . . .	157
7.2.2 REPEAT . . . . .	159
7.3 Consejos y tips . . . . .	159
<b>III Listas y funciones</b>	<b>161</b>
<b>8 Tipos de datos III: listas/factores</b>	<b>163</b>
8.1 Listas . . . . .	163
8.2 Factores . . . . .	177
8.3 Fechas y horas . . . . .	182

8.4	Consejos y tips . . . . .	188
8.5	□ Ejercicios . . . . .	190
<b>9</b>	<b>Creación de funciones</b>	<b>195</b>
9.1	Primera función . . . . .	196
9.2	Segunda función . . . . .	200
9.3	Variables locales/globales . . . . .	205
9.4	□ Ejercicios . . . . .	207
<b>10</b>	<b>¿Qué sabemos hacer?</b>	<b>211</b>
10.1	Incursión aleatoria . . . . .	213
10.1.1	Pseudoaleatoriedad . . . . .	217
10.2	Recursos . . . . .	219

## **List of Tables**



# List of Figures

1.1	Pantalla inicial de la plataforma CRAN de R. . . . .	21
1.2	Pantalla de instalación de R en Mac OS. . . . .	22
1.3	Pantalla previa de instalación de R en Windows. . . . .	22
1.4	Pantalla final de instalación de R en Windows. . . . .	23
1.5	Primera pantalla al abrir el ejecutable de R. . . . .	23
1.6	Primera suma en la consola de R. . . . .	24
1.7	Descargar el ejecutable de RStudio para su posterior instalación. . .	25
1.8	Primer recibimiento de nuestro mejor amigo RStudio. . . . .	26
1.9	Lanzando a consola nuestras primeras órdenes en RStudio. . . . .	26
1.10	Environment de variables. . . . .	27
1.11	Panel multiusos. . . . .	28
1.12	Abriendo nuestro primer script de R. . . . .	28
1.13	Escribiendo y guardando nuestro primer script. . . . .	29
1.14	Paquetes disponibles en R. . . . .	30
1.15	Ejemplo de que la orden lanzada ha acabado. . . . .	32
1.16	Menú de opciones de nuestro editor . . . . .	32

1.17 Personalizar el color de fondo de nuestro editor, la letra y el tamaño de fuente . . . . .	33
2.1 Excel en una noche loca. . . . .	40
2.2 Cuando Excel dice basta. . . . .	40
2.3 Los centenarios con biberón. . . . .	41
2.4 Lanzando a consola nuestras primeras órdenes en RStudio. . . . .	42
2.5 Environment. . . . .	43
2.6 Panel de ayuda. . . . .	46
3.1 Paquete stringr para manejar cadenas de texto más complejas . . . . .	61
4.1 Crear un nuevo proyecto en R. . . . .	88
4.2 Opciones de creación. . . . .	89
4.3 Clickar en «New project». . . . .	89
4.4 Nombre del proyecto. . . . .	90
4.5 Abrir nuestro primer script de R. . . . .	90
4.6 Descripción al inicio del código. . . . .	91
4.7 Guardamos el código. . . . .	91
4.8 Guardamos el código. . . . .	91
4.9 Saltar de proyecto en proyecto. . . . .	92
4.10 Subcarpeta «CODIGOS». . . . .	92
4.11 Creamos un fichero «variables.R». . . . .	93
4.12 Escribimos una serie de variables fijas para luego ser usadas. . . . .	93
4.13 Consultar directorio de trabajo predeterminado y archivos contenidos en él. . . . .	94

<i>LIST OF FIGURES</i>	11
4.14 Fijamos de forma automática el directorio de trabajo. . . . .	95
4.15 Cargar archivos de nuestro directorio de trabajo. . . . .	96
4.16 Cargar archivos de nuestro directorio de trabajo. . . . .	96
4.17 Guardamos con la casilla «source on save» activada para que además de guardar se ejecute el código. . . . .	97
4.18 Cálculos con las variables definidas: suma, concatenación de texto y diferencia de fechas. . . . .	98
5.1 Menú desplegable de variables (columnas) de un data.frame. . . . .	116
5.2 Creando nuestro primer data.frame en el script. . . . .	116
5.3 Llamando a nuestro script desde nuestro código principal. . . . .	118
5.4 Instalamos y cargamos los paquetes necesarios al principio de nues- tro main.R. . . . .	118
5.5 Menú desplegable con los data.frame de prueba en datasets . . . . .	119
5.6 Menú desplegable con los data.frame de prueba en datasets . . . . .	119
6.1 Importación de ficheros de extensión .RData. . . . .	134
6.2 Archivos de la pandemia en el ISCIII. . . . .	142
6.3 Paquete readr. . . . .	145
6.4 Paquete tidyR. . . . .	145
6.5 Secciones en el código. . . . .	146
6.6 Líneas de código. . . . .	146
6.7 Margen derecho. . . . .	147
8.1 Paquete lubridate. . . . .	189
8.2 Paquete lubridate. . . . .	189

10.1 Cuando intentas aprender todos los paquetes. . . . .	212
---	-----

# Prefacio

Este manual ha sido diseñado para la asignatura de Descripción y Exploración de Datos del grado de Estadística Aplicada (UCM, curso académico 2021-2022), y está elaborado por [Javier Álvarez Liébana](#).

Dicho manual ha sido elaborado a su vez en R con [{bookdown}](#). Puedes ver un resumen de las funcionalidades algunos paquetes documentados por el equipo de [R Studio](#) en sus [esquemas resumen](#). El **código** de dicho manual se encuentra en GitHub ([https://github.com/dadosdelaplace/cursoR\\_intro\\_2021\\_2022](https://github.com/dadosdelaplace/cursoR_intro_2021_2022)).

Para **elaborar informes o libros** con una estructura similar, de forma nativa en R, el paquete `{bookdown}` puede ser instalado desde la plataforma CRAN o desde su versión en desarrollo actualizada en Github:

```
install.packages("bookdown")
# o desde su versión en desarrollo actualizada
# devtools::install_github("rstudio/bookdown")
```

## Propósito

El **objetivo** de este tutorial es introducir a la programación y análisis estadístico en R a toda aquella persona que nunca se haya iniciado en él, **sin necesitar conocimientos previos** de programación (aunque siempre ayuda, obviamente). Con este manual no se pretende que adquieras un vasto y experto conocimiento de R, pero si lo suficiente como para lograr **5 objetivos**:

- **No tener miedo** a programar.
- Ser capaces de abordar desde la programación (ya sea en R o en otro lenguaje) pequeños problemas, con el fin de saber **conceptualizarlos**.
- Entender los **usos y costumbres del lenguaje R**
- Darte pinceladas de sus posibilidades que te sirvan de trampolín para ir investigando por tu cuenta en el campo en el que lo vayas a aplicar.
- Algunos **trucos sencillos** para que el trabajo sea más rápido, tanto en tiempo de escritura como de ejecución.
- Hacer **enfasis en la importancia de la visualización de datos** en estadística.

## Requisitos

- **Conexión a internet:** se necesitará tener una conexión a internet disponible para la descarga de algunos datos y paquetes.
- **Instalar R** (ver 1). **R será nuestro lenguaje**, nuestro diccionario, nuestro castellano, nuestra ortografía para poder «comunicarnos» con el ordenador.  
<https://cran.r-project.org/>
- **Instalar RStudio** (ver 1.1). De la misma manera que podemos escribir el

el mismo texto en castellano en una tablet, en un ordenador, en un Word, en un papel o en un tuit, en programación podemos usar distintos IDE (**entornos de desarrollo integrados**, nuestro Office), para que el trabajo sea más cómodo. Nosotros trabajaremos con RStudio.

Todo lo necesario para seguir este curso es de **descarga gratuita**: viva el software libre, abajo Excel.

## Emojis

 **Info:** siempre que veas el **ícono de la bombilla** encontrarás consejos o tips para ampliar y facilitar tu programación. Además en cada **cajita de código**, si pasas el ratón, encontrarás un botón en la esquina superior derecha de la caja para copiar el código directamente a tu consola.

- **Ejercicios:** siempre que veas el **ícono de un documento escrito** encontrarás ejercicios con soluciones para que vayas afianzando conceptos (una de las soluciones, normalmente siempre habrá muchas formas distintas de realizar la misma tarea, prioriza la más sencilla y limpia).
  
- □ **Copiar código:** siempre que veas el **ícono de dos documentos** en las **cajas de código** (aparecen al pasar el ratón), podrás hacer click para copiar directamente el código que haya dentro.

- **Glosario:** siempre que veas la pila de libros encontrarás una **sección de glosario con algunos términos estadísticos y conceptos básicos.**

## Sobre el autor

Esto de presentarse a sí mismo es siempre un poco raro pero vamos a intentarlo.

Mi nombre es **Javier Álvarez Liébana**, soy **matemático**, nacido en 1989 en **Carabanchel (Madrid)**, pasando por Bologna (Italia). Tras terminar licenciatura y Máster en Ingeniería Matemática, recibí en julio de 2018 el título de **Doctor en Estadística** (por la Universidad de Granada, con dos estancias en Université Pierre et Marie Curie)

Además de **investigador** (con plaza y acreditación de Ayudante Doctor en la Facultad de Estudios Estadísticos de la Universidad Complutense de Madrid, tras ocupar dicha plaza en la Universidad de Oviedo), soy **docente** en dicha facultad y ando intentando eso de la **divulgación en estadística y dataviz** (visualización de datos) en redes sociales

- [Twitter](#)
- [Instagram](#)

## Licencia



Este documento es publicado bajo **licencia pública general GNU**, una licencia libre de copyleft que garantiza a los usuarios finales (personas, organizaciones, compañías) la **libertad de usar, estudiar, compartir (copiar) y modificar el software**,

**citando adecuadamente al autor del mismo.**



## **Part I**

### **Toma de contacto**



# Chapter 1

## Instalación

Vamos a necesitar solo 3 pasos (y conexión a internet).

- **Paso 1:** entra en la web <https://cran.r-project.org/> y en la pantalla de inicio selecciona la instalación acorde a tu sistema operativo (ver imagen 1.1)

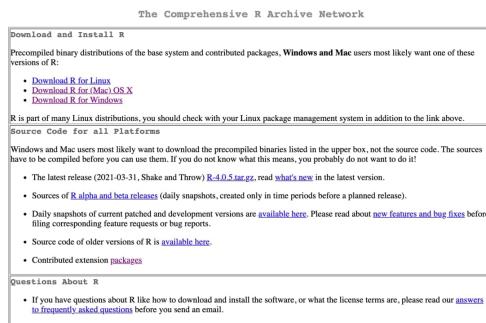


Figure 1.1: Pantalla inicial de la plataforma CRAN de R.

- **Paso 2:** para sistemas operativos Mac basta con que hacer click en el archivo .pkg, y abrirlo una vez descargado (ver imagen 1.2)

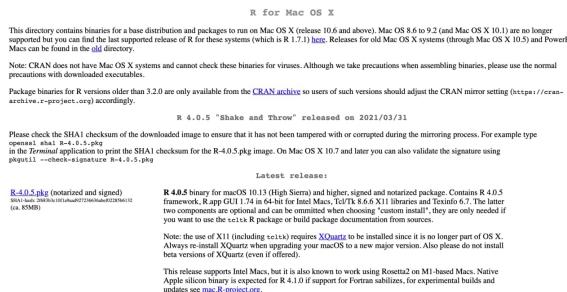


Figure 1.2: Pantalla de instalación de R en Mac OS.

Para sistemas operativos Windows, debemos clickar en `install R for the first time` (ver imagen 1.3) y en la siguiente pantalla hacer click en `Download R for Windows` (ver imagen 1.4). Una vez descargado, abrirlo como cualquier archivo.

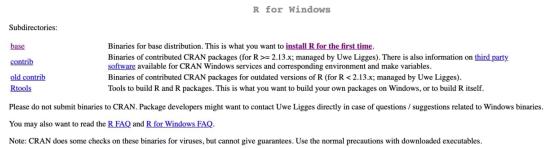


Figure 1.3: Pantalla previa de instalación de R en Windows.

- **Paso 3:** tras su instalación tendrás en tu escritorio (Windows) o en tu Launchpad (Mac Os) un ejecutable de R para abrir. En Windows puede que tengas dos ejecutables i386 y x64 (como todo programa en Windows está la versión de 32 y de 64 bits, haz click preferiblemente - si lo tienes - en el de x64). Te saldrá algo parecido a lo que observas en la imagen 1.5.

Para comprobar que está correctamente instalado, prueba a escribir en la consola el siguiente código (**recuerda:** los códigos puedes copiarlos directamente de la

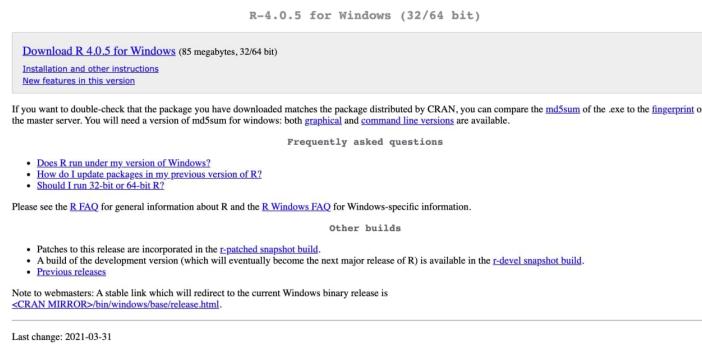


Figure 1.4: Pantalla final de instalación de R en Windows.

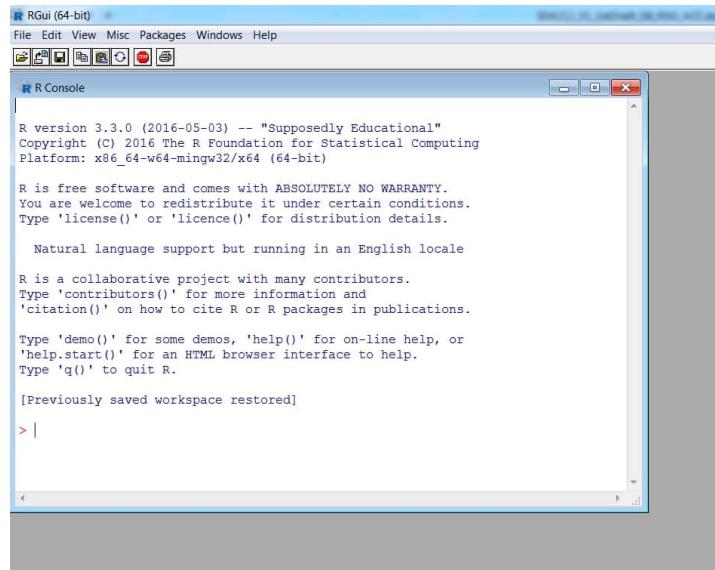


Figure 1.5: Primera pantalla al abrir el ejecutable de R.

cajita en la que está haciendo click en el botón de la esquina superior derecha)

```
a <- 1
b <- 2
a + b
```

```
## [1] 3
```

```
R Console
Type 'license()' or 'licence()' for distribution details.
Natural language support but running in an English locale
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

During startup - Warning messages:
1: Setting LC_CTYPE failed, using "C"
2: Setting LC_NUMERIC failed, using "C"
3: Setting LC_TIME failed, using "C"
4: Setting LC_MESSAGES failed, using "C"
5: Setting LC_MONETARY failed, using "C"
[R.app GUI 1.70 (7735) x86_64-apple-darwin15.6.0]

WARNING: You're using a non-UTF8 locale, therefore only ASCII characters will work.
Please read R for Mac OS X FAQ (see Help) section 9 and adjust your system preferences accordingly.
[History restored from /Users/javierolvarezlebana/.Rapp.history]

> a <- 1
> b <- 2
> a + b
[1] 3
```

Figure 1.6: Primera suma en la consola de R.

**¡Enhorabuena!** Si te ha devuelto [1] 3, ya has hecho más de lo que parece: has definido dos variables a y b, has **asignado un valor numérico a cada variable** y las hemos usado. **Ya sabemos usar R como calculadora.**

**Asignación:** como habrás advertido, en R usaremos <- para asignar valores en lugar de =, como una flecha.

**Línea de consola:** como habrás advertido, en R usaremos <- para asignar valores en lugar de =, como una flecha.

Bonita esta interfaz no es, así que la cerraremos y no la abriremos más. Tenemos nuestro lenguaje instalado, vamos a **instalar nuestro Word** para poder programar de forma cómoda.

## 1.1 Instalación de RStudio

Para instalar RStudio deberemos ir a la web <https://www.rstudio.com/products/rstudio/download/#download> y seleccionar el ejecutable que te aparezca acorde a tu sistema operativo (ver imagen 1.7). Tras descargar el ejecutable, hay que abrirlo como otro cualquier otro ejecutable y dejar que termine la instalación.

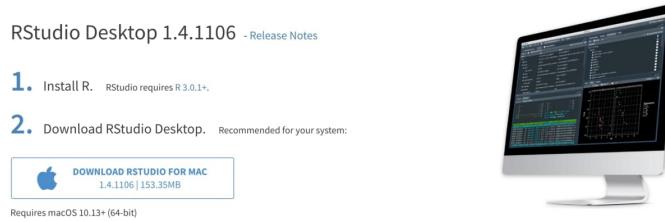


Figure 1.7: Descargar el ejecutable de RStudio para su posterior instalación.

Tras instalar tendremos en el escritorio o Launchpad un ejecutable de RStudio que abriremos. Se nos aparecerá una pantalla similar a esta:

- **Consola:** es el nombre para llamar a esa ventana grande que te ocupa la mayor parte de tu pantalla. Prueba a escribir el mismo código que antes en ella (es el equivalente a la consola de R que hemos abierto al principio).

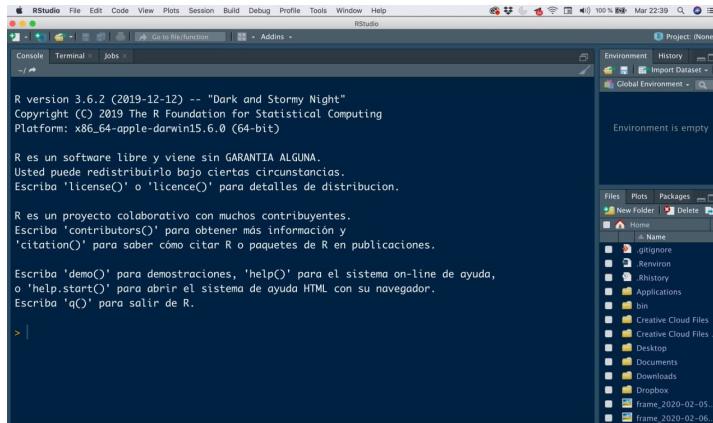


Figure 1.8: Primer recibimiento de nuestro mejor amigo RStudio.

```
a <- 1
b <- 2
a + b
```

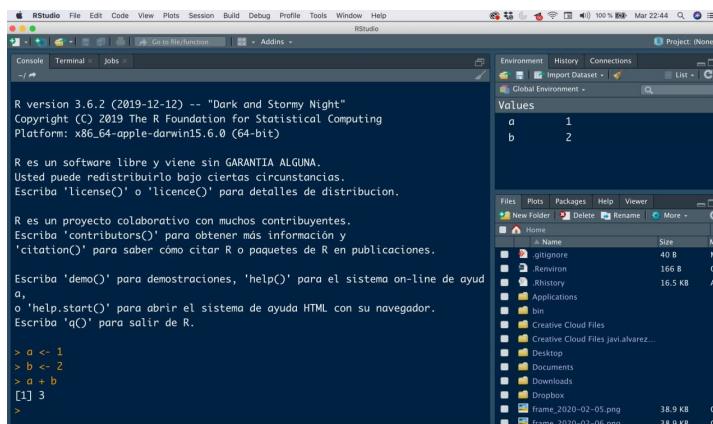


Figure 1.9: Lanzando a consola nuestras primeras órdenes en RStudio.

### La consola será donde ejecutaremos órdenes y mostraremos resultados

- **Environment (entorno):** la pantalla pequeña (puedes ajustar los márgenes con el ratón a tu gusto) que tenemos en la parte superior derecha se denom-

ina environment o entorno de variables, donde como puedes ver, tras ejecutar el pequeño código en la consola, nos informa de que tenemos dos variables numéricas y su valor asignado. Nos **mostrará las variables que tenemos definidas, el tipo y su valor.**

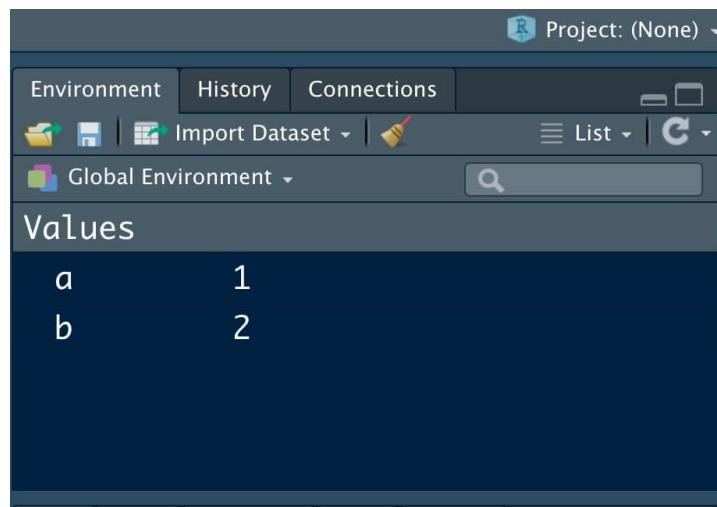


Figure 1.10: Environment de variables.

- **Panel multiusos:** la ventana que tenemos en la parte inferior derecha no servirá para buscar ayuda de comandos y órdenes, además de para visualizar gráficos. Lo veremos cuando sea necesario.

### WTF ¿Y DÓNDE PROGRAMAMOS?

**¿Estás emocionado/a? Vamos a abrir nuestro primer script** (script = documento en el que programamos, nuestro .doc, pero aquí será un archivo .R).

Haz click en el menú superior en `File << New File << R Script` como se muestra en la imagen 1.12

Tras abrirla tendremos una cuarta ventana: esta será la ventana de nuestros códigos.

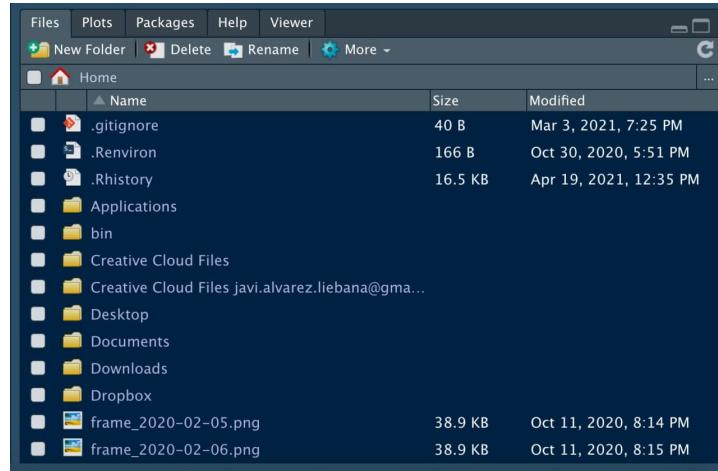


Figure 1.11: Panel multiusos.

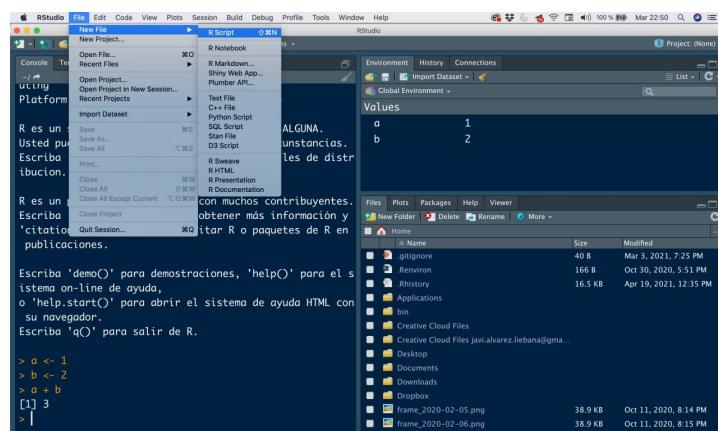


Figure 1.12: Abriendo nuestro primer script de R.

gos, la ventana más importante ya que es donde **escribiremos lo que queremos ejecutar**. Escribe el código de arriba en ese script y guarda el archivo haciendo click en el botón Save current document

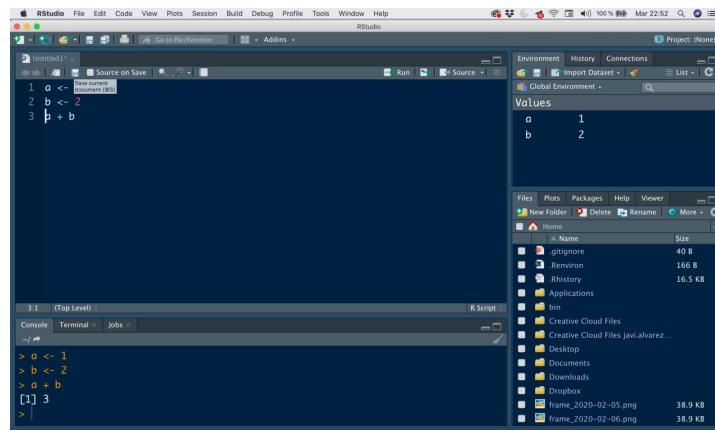


Figure 1.13: Escribiendo y guardando nuestro primer script.

Ese código no se ejecuta salvo que se lo digamos. Tenemos dos opciones para ello: o copiamos el trozo de código que queramos y lo pegamos en la consola (como hemos hecho al principio), o **activamos el cuadrado Source on save a la derecha del botón de guardar** y volvemos a hacer click en el botón de guardar: siempre que esa opción esté activada, al guardar no solo se nos guarda el archivo .R sino que además se ejecuta solo y nos devuelve los resultados por consola.

Listo, tienes instalado (casi) todo correctamente.

## 1.2 Instalación de paquetes

El lenguaje R tiene 3 ventajas principales:

- Es un lenguaje creado por y para estadísticos/as, por lo que está pensado

para optimizar al máximo los recursos, y poder hacer un análisis estadístico de calidad

- Es **software libre** (como C, C++, Python, Fortran, y otros tantos lenguajes). El software libre no solo tiene una ventaja evidente (es gratis, ok) en su instalación sino que permite acceder al código en el que están programados los comandos y permite hacer uso de trozos de código de otras personas.
- Es un **lenguaje modular**: en la instalación no se instalan todas las funcionalidades sino que instala un mínimo para poder funcionar, de forma que se ahorra espacio en disco y en memoria. Al ser software libre, existen trozos de código hechos por otras personas llamados **paquetes**, que podemos ir instalando a nuestro gusto según los vayamos necesitando. Esto es una ventaja enorme ya que R tiene una comunidad de usuarios gigante, con **más de 17 000 paquetes**: ¡hay más de 17 000 trozos de código validados por la comunidad y la plataforma, de forma gratuita!

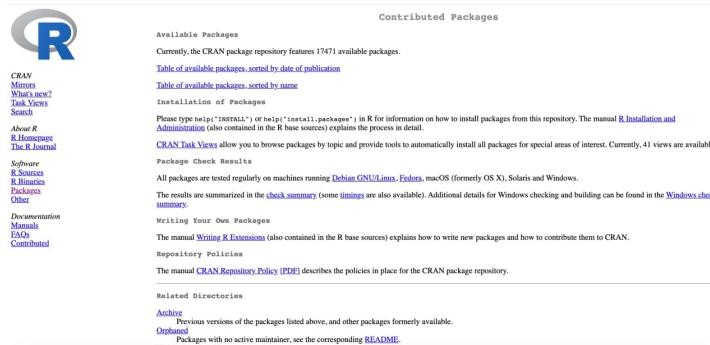


Figure 1.14: Paquetes disponibles en R.

Esto nos ahorra muchísimo tiempo ya el **90% de lo que querramos hacer ya lo habrá querido hacer otra persona y podemos usar o adaptar su código** para no

empezar de cero. Vamos a instalar un paquete gráfico (`{ggplot2}`) que necesitaremos. Para ello, escribe en tu consola el siguiente código y pulsa enter.

```
install.packages("ggplot2")
```

Dicha orden (puede tardar un poco la primera vez, depende de tu conexión a internet) lo que hará será acceder a la web de R, bajarse a tu ordenador los trozos de código incluidos en el paquete llamado `{ggplot2}` (para realizar gráficas), y dejarlos para siempre en él: **la instalación de paquetes SOLO ES NECESARIO la primera vez** que se usa dicho paquete en la vida del ordenador, no hace falta hacerlo cada vez que lo usas.

Una vez que tenemos los trozos de código (el paquete) en nuestro ordenador, en cada sesión de R que abramos (cada vez que cierres y abras RStudio) deberemos (si queremos) llamar a ese paquete que tenemos instalado, escribiendo el siguiente comando en consola

```
library(ggplot2)
```

Welcome to software libre

## 1.3 Consejos y tips

¿Cómo saber cuando la orden lanzada en consola ha terminado?

A veces R y RStudio son tan silenciosos que no sabemos si ha acabado la orden que acabamos de lanzar en la consola o no.

 **Truco:** siempre que veas este símbolo > como última línea en la consola significa que está listo para que le escribamos otra orden (es la forma cariñosa de

decirte que ya ha acabado, ver imagen 1.15)

```
> install.packages("plotly")
probando la URL 'https://cran.rstudio.com/bin/macosx/el-capitan/contrib/3.6/plotly_4.9.3.tgz'
Content type 'application/x-gzip' length 3063927 bytes (2.9 MB)
=====
downloaded 2.9 MB

The downloaded binary packages are in
  /var/folders/kj/rf2k72b528n_lxsj60f76j2w0000gn/T//RtmpgfgW0n/downloaded_packages
>
```

Figure 1.15: Ejemplo de que la orden lanzada ha acabado.

¿Cómo prevenir la fatiga visual programando?

Estar delante de una pantalla de ordenador, con la vista muy fija mientras se programa, puede que acabes teniendo cierta fatiga visual en el trabajo.

 **Truco:** te aconsejo que cambies en tu RStudio la tonalidad del fondo de tu programa, en tonos oscuros y no blancos (¿te has fijado que mis capturas tienen un azul cobalto oscuro de fondo mientras el tuyo es un blanco nuclear? Echa un vistazo las imágenes 1.16 y 1.17)

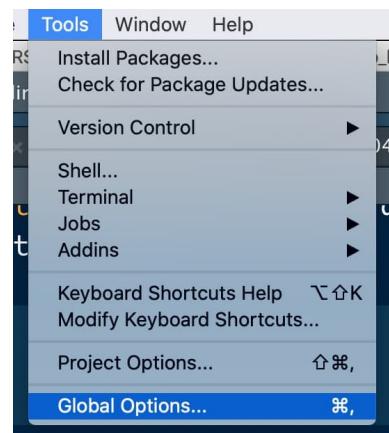


Figure 1.16: Menú de opciones de nuestro editor

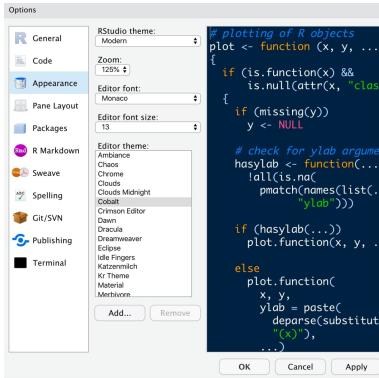


Figure 1.17: Personalizar el color de fondo de nuestro editor, la letra y el tamaño de fuente

### Entender los paquetes

Haciendo una metáfora con la colección de libros que tengas en casa: con la **instalación hemos comprado el libro** y lo tenemos en nuestra estantería (para siempre), con la llamada al paquete, por ejemplo `library(ggplot2)`, lo que hacemos es decidir, de entre todos los libros de la estantería, cuales queremos llevarnos de viaje (en cada maleta que hagamos).

Los **paquetes usados** los verás denotados como `{nombre_paquete}` a lo largo del manual.



## Chapter 2

# Primeros pasos

### ¿Empezamos?

Veamos antes **un poco de historia** sobre nuestro deidad.

Allá por 1975, los **laboratorios Bell** (los que inventaron la radio moderna tal y como la conocemos), necesitaban una alternativa a los lenguajes más «rudos» y antiguos como C++ o Fortran, lenguajes rápidos en la ejecución pero complejos en su uso, con una gran curva de aprendizaje y con **muy poca capacidad en la visualización de datos** que se empezaba a necesitar.

Así que en **1976 sacaron la primera versión del lenguaje conocido s** (hasta entonces estaba de moda lo de llamar a los lenguajes con una sola inicial). En **1980** se empezó a distribuir la primera versión pública de **s**, más allá de los laboratorios Bell, y en **1988** se añadieron bastantes funcionalidades nuevas como poder aplicar funciones a otras funciones (los famosos apply() que ya veremos). Años más tarde, en **1991**, dicho lenguaje se simplificó, reescribiendo muchas subrutinas

de otros lenguajes más primitivos, para tener una versión muy parecida al actual R, permitiendo el uso de operadores, `data.frames` (que veremos) y otro tipo de objetos, sencillos en la programación pero muy versátiles.

Sin embargo, salvo uso docente, S tenía licencia así que **en 1992 Ross Ihaka y Robert Gentleman se lanzaron a crear una versión de S libre y gratuita**, un trabajo de casi 8 años hasta que en el año 2000, ambos investigadores de la Universidad de Auckland en Nueva Zelanda lanzaron la primera versión estable del lenguaje.

Tras dicho lanzamiento, se creó un **equipo de expertos en estadística computacional (el conocido como R Development Core Team)** que es el que se encarga de mantener toda la arquitectura de R y los que se encargan de actualizar y mejorar el paquete `{base}`, una **librería motor** sobre la que se construye el resto de funciones.

## 2.1 Excel no es tu amigo

R es un lenguaje de programación, de alto nivel para el usuario y modular. Los lenguajes de alto nivel como R, Python (curso interactivo de Python en <https://checkio.org/>) o Matlab, facilitan la programación al usuario, teniendo que preocupa parte solo de la tarea de programar. Son lenguajes con una menor curva de aprendizaje aunque suelen ser más lentos en su ejecución en comparación con lenguajes de bajo nivel (C, C++ o Fortran), lenguajes muy rápidos en su ejecución pero cuya programación requiere un mayor tiempo y formación, teniendo que además estar pendiente del tipo de variables, espacio en memoria, etc.

Por su arquitectura, R es un lenguaje que puede ser usado para un propósito general pero que está especialmente diseñado para el **análisis estadístico de datos**. Su

**modularidad** nos da la ventaja de que podemos instalar las funcionalidades que vayamos necesitando de forma progresiva.

### ¿Por qué no es recomendable usar Excel?

- **Software de pago:** Excel, al igual que el resto de programas de Microsoft o SPSS (por desgracia programa estrella de nuestro sistema sanitario), es un programa de pago. A nivel individual, todos hemos tenido una versión que no hemos pagado, pero dicha evasión no se la puede permitir una empresa o administración, que debe de pagar altas cantidades de dinero anuales por las licencias, dinero que no sería necesario si los investigadores y trabajadores tuvieran formación (remunerada) en otras herramientas de software libre.
- **Software cerrado:** no solo es de pago sino que es cerrado, así que solo podemos hacer lo que Excel ha creído que interesante que podamos hacer. Incluso con la programación de MACROS, las funcionalidades de Excel siguen siendo mucho más limitadas ya que viene «programadas» de antemano
- **Alto consumo de memoria:** dicha programación predeterminada hace que Excel ocupe muchísimo espacio en el disco duro y tenga un alto consumo de memoria (la memoria es lo que te permite hacer varias tareas a la vez en tu ordenador).
- **No es universal:** no solo es de pago sino que además, dependiendo de la versión que tengas de Excel, tendrá un formato distinto para datos como fechas, teniendo incluso extensiones distintas, de forma que un archivo .xls abierto por un Excel moderno puede provocar errores en la carga.

- **¡ES SOLO UNA HOJA DE CÁLCULO!**: el propio Microsoft desaconseja el uso de Excel para el análisis de grandes volúmenes de datos. El Excel es una herramienta maravillosa para ser usada como una sencilla hoja de cálculo: llevar las cuentas de tu familia, de tu pequeño negocio, una declaración de la Renta sencilla, planificar viajes, etc. Pero el programa **NO ESTÁ DISEÑADO** para ser una base de datos ni para análisis detallado, y muchos menos pensado para generar un entorno flexible para el análisis estadístico y la visualización de datos.

**¿Puedes ser el mejor partiendo un filete con una cuchara?** Seguramente puedes (en Excel puedes hasta programar con macros), y si siempre lo hiciste así, acabarás normalizándolo, pero seguirás siendo una persona comiendo filete con cuchara.



### ¿Qué sucede si usamos la herramienta equivocada?

Tres ejemplos:

- **Problemas para codificar fechas:** en 2016 se publicó una revisión de artículos en genética, descubriendo que 1 de cada 5 artículos contenían errores debido a una mala codificación de las fechas, convirtiendo por ejemplo los genes Septin-2 (conocido como SEPT2) en fechas, y al revés ([Ziemann et al., 2016](#)).



Figure 2.1: Excel en una noche loca.

- **Problemas de memoria:** un Excel permite por defecto una cantidad máxima de filas. Aunque dicha cantidad se puede ampliar, sigue siendo finita, por lo que cuando superas el umbral de filas, al añadir filas Excel te borra registros **sin avisarte de que lo está haciendo**. Esto es lo que sucedió con los [registros de casos covid en Reino Unido](#).



Figure 2.2: Cuando Excel dice basta.

- **Problemas para codificar edades:** una variable de tipo fecha, aunque nosotros la veamos con letras, en realidad es una variable numérica que representa los días que han pasado desde una fecha origen. En función de las distintas versiones de Excel, dicha fecha origen cambia. Además,

si se codifica mal la fecha en formato dd-mm-YY, dicho formato cuando se exporta a otro excel en texto, no permite distinguir a un nacido en 1918 y a un nacido en 2018, así que podemos estar confundiendo personas de 103 años con niños de 3 años (y es lo que [sucedió en España](#), observando unas tasas de mortalidad en niños muy pequeños equivalentes a personas mayores).

Sanidad admite un error en la mortalidad infantil por covid: “Cuentan centenarios como menores”



Figure 2.3: Los centenarios con biberón.

## 2.2 Primeros pasos: calculadora

¿Te acuerdas de lo que era la **consola**? Vamos a trabajar de momento en esa ventana que tienes en la ventana inferior.

Lo que ya hemos descubierto en los pasos de la instalación (ver imagen 2.4) es que la consola de R tiene una función muy básica y evidente: nos sirve de calculadora. Un ejemplo muy simple: si escribimos 3 en la consola y pulsamos *ENTER*, la consola nos mostrará el resultado de la suma

```
1 + 2
```

```
## [1] 3
```

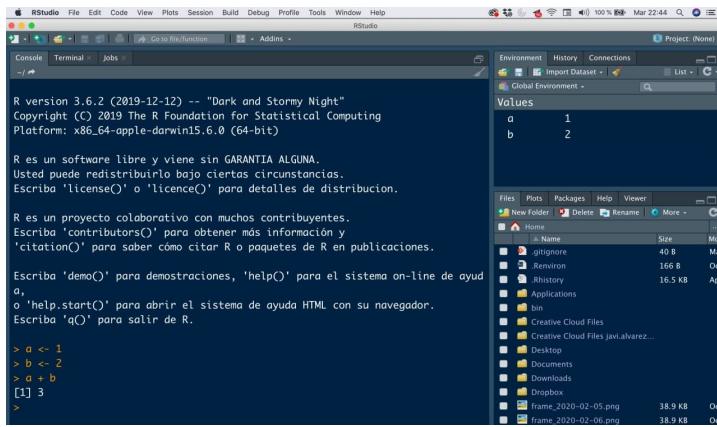


Figure 2.4: Lanzando a consola nuestras primeras órdenes en RStudio.

### ¿Pero cuál es la diferencia entre una calculadora y un lenguaje de programación?

Imagina que dicha suma 3 la quisiéramos utilizar para un segundo cálculo: ¿y si en lugar de lanzarlo a la consola sin más lo **almacenamos en alguna variable**?

Como hemos visto en la instalación de RStudio, para **asignar variables** lo haremos con la orden `x <- 1 + 2`: una variable de nombre `x` va a tener asignada `<-` lo que valga la suma `1 + 2`

```
x <- 1 + 2
```

Como puedes comprobar, en tu parte superior derecha (nuestro entorno de variables), podrás ver como una nueva variable `x` es ahora visualizada, con su valor asignado (**no se mostrará en consola** salvo que escribas 3 en ella: R asume que no querías visualizarla en consola sino solo guardarla).

Dicha variable `x` además podemos reciclarla para definir una variable `y`, restándole una constante.

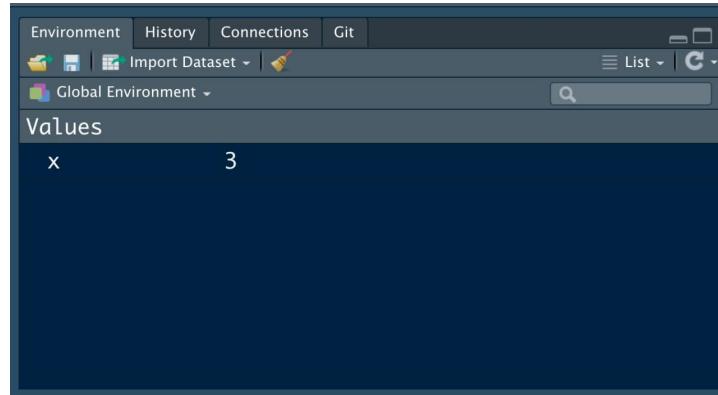


Figure 2.5: Environment.

De la misma manera que hemos hecho restas, sumas y multiplicaciones, R tiene todas las operaciones clásicas que podrías tener en una calculadora. Prueba a ejecutar en la consola las órdenes `x^2`, `sqrt(x)` o `abs(y)`: R calculará las operaciones *elevar al cuadrado*, *raíz cuadrada* y *valor absoluto* de la variable que tengan entre paréntesis

```
x^2
```

```
## [1] 9
```

```
sqrt(x)
```

```
## [1] 1.732051
```

```
y <- x - 5
```

```
abs(y)
```

```
## [1] 2
```

## 2.3 Tipos de errores

Durante tu aprendizaje en R va a ser **muy habitual** que las cosas no salgan a la primera, **apareciendo en consola mensajes en un color rojo**. Un **miedo** muy habitual cuando se empieza a programar es pensar que si haces algo mal o aparece algún mensaje de error, el ordenador puede explotar en cualquier momento. **A programar se aprende programando**, así que haz las pruebas que quieras, lo peor que puede pasar es que necesites cerrar sesión en R Studio y abrirlo de nuevo.

Dado que el 99.99999% de veces tu código tendrá errores que deberás ir solventando, no está de más conocer los tipos de mensajes que R puede sacarte por consola:

- **Errores:** los mensajes de error irán precedidos de la frase «**Error in...**», dándose a veces incluso el tipo de error y la línea de código en la que se ha producido. Veamos un ejemplo intentando sumar un número a una cadena de texto.

```
"a" + 1
```

```
## Error in "a" + 1: argumento no-numérico para operador binario
```

Los **errores son aquellos fallos que seguramente impidan la ejecución** del código. Un **error muy habitual** es intentar acceder a alguna función de algún paquete que, o bien no tenemos instalado, o bien no hemos llamado haciendo uso del `library()`: estás intentando leer un libro de tu biblioteca pero ni siquiera has ido a la tienda a «comprarlos».

- **Warnings:** los mensajes de *warning* irán precedidos de la frase «**Warning:...**», y son los fallos más delicados ya que son **posibles errores o incoherencias** que R detecta en tu código pero que **no van a hacer que tu código deje de ejecutarse**, aunque probablemente no lo haga como a ti te gustaría. Un ejemplo es cuando tratamos de hacer la raíz cuadrada de un número negativo.

```
sqrt(-1)
```

```
## Warning in sqrt(-1): Se han producido NaNs  
## [1] NaN
```

¿Ha ejecutado la orden? Sí, pero te *advierte* de que el resultado de la operación es un NaN, un valor que no existe (al menos dentro de los números reales), un *Not A Number* (ver Sección 3.6).

- **Mensajes de control:** los mensajes de control serán aquellos que aparecerán por consola sin empezar por «Error in...» ni «Warning:...». Dichos mensajes, que puedes incluir tú mismo en tu código con funciones como `cat()` para monitorizar la ejecución de códigos largos, no son errores ni problemas, son simplemente información que R considera útil aportarte.

## 2.4 Consejos y tips

### Argumentos de una función

Las órdenes `sqrt(x)` y `abs(y)` se llaman **funciones**, y la variable que tienen entre paréntesis se llama **argumento de la función**: una variable que toma de entrada

la función y con la que opera internamente.

### Panel de ayuda

Si escribes en la consola `? nombre_funcion` (por ejemplo, escribe en la consola `? sqrt`), en el panel inferior derecho te aparecerá una **documentación de ayuda** de la función para saber que argumentos necesita, como puedes usar la función, qué es lo que te devuelve, ejemplos de uso, etc.

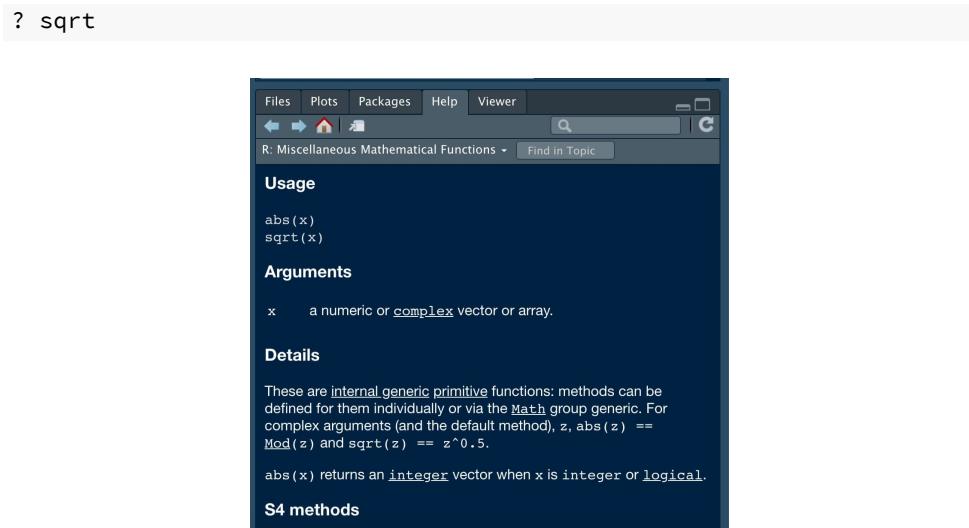


Figure 2.6: Panel de ayuda.

## 2.5 □ Ejercicios

- Ejercicio 1: calcula en consola la suma de 3 más 4, y todo ello multiplicado por 10, y asígnalo a una variable `x`.

- Solución:

```
x <- (3 + 4) * 10
```

- Ejercicio 2: usando la variable x ya definida, calcula  $x - 5$  y guárdalo en una nueva variable y.

- Solución:

```
y <- x - 5
```

```
y
```

```
## [1] 65
```

- Ejercicio 3: usando las variables x e y ya definidas, calcula la raíz cuadrada del máximo entre ambas, y guárdalo en una nueva variable z.

- Solución:

```
z <- sqrt(max(x, y)) # No hace falta gastar una línea por cada orden (cada asignación que hagas es  
z
```

```
## [1] 8.3666
```



## **Part II**

# **Conceptos básicos**



# Chapter 3

## Tipos de datos I: vectores

Bien, ya controlamos la calculadora. Vamos a ir más allá: ¿y si en lugar de tener un solo número tenemos un **CONJUNTO de elementos**? En este capítulo vamos a ver un clásico de cualquier lenguaje de programación: los **vectores o arrays**.

### 3.1 Vectores numéricos

Un conjunto de elementos del mismo tipo se llama **vector** (en este caso de números), y de hecho un número individual (por ejemplo, 1) es en realidad un vector de longitud uno (un solo elemento).

La forma más sencilla de **crear un vector** en R es con el comando `c()` (de **concatenar elementos**), y basta con introducir sus elementos entre paréntesis, y separados por comas. Vamos a crear el vector con los tres primeros números naturales pares (el 0 no es natural, no seas bárbaro/a).

```
z <- c(2, 4, 6)
```

```
z
```

```
## [1] 2 4 6
```

Como ves ahora en el **environment tenemos una colección de elementos**, tres en concreto, guardados en una misma variable. La longitud de un vector se puede calcular con el comando `length()`.

```
length(z)
```

```
## [1] 3
```

Además podemos **concatenar a su vez vectores**: vamos a concatenar el vector `z` consigo mismo, y añadiéndole al final un 8.

```
c(z, z, 8)
```

```
## [1] 2 4 6 2 4 6 8
```

La última concatenación lo que nos ha dado son los tres primeros pares, después de nuevo los tres primeros pares, y por último un 8.

### 3.1.1 Secuencias con un patrón

Muchas veces nos gustaría **crear vectores de una forma mucho más rápida**, por ejemplo, para tener un vector de índices que queramos recorrer. Supongamos que queremos el vector de los primeros 21 números naturales. Si construyéramos el vector como antes, tendríamos ejecutar el comando `c(1, 2, 3, 4, 5, ...)` hasta el número 21. ¿Un poco largo, no?

El comando `seq()` nos permite crear una **secuencia desde un elemento inicial hasta un elemento final, avanzando de uno en uno.**

```
seq(1, 21) # secuencia desde 1 hasta 21 de uno en uno
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

Es importante que no perdamos el foco de que **programar es similar a escribir en un idioma**, por lo que si hay algo que se puede decir de una forma más limpia y que se entienda mejor, ¿por qué no usarlo? Siempre que queramos definir secuencias entre dos números naturales (por ejemplo, entre 1 y un valor  $n$ ), cuya distancia entre elementos consecutivos sea uno, el comando `1:n` nos devuelve lo mismo que la orden `seq(1, n)`. Además, si el elemento inicial es mayor que el final, R entenderá solo que la secuencia la queremos decreciente.

```
n <- 21
```

```
1:n # secuencia desde 1 hasta n (21) de uno en uno
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
```

```
17:1 # secuencia decreciente de 17 a 1
```

```
## [1] 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

También podemos definir **otro tipo de distancia entre dos elementos consecutivos** (conocido como **paso de discretización**), por ejemplo de 0.5 en 0.5, o bien definir una secuencia entre un valor inicial y un valor final con un número de elementos fijo (y que sea R el que decida la distancia entre elementos consecutivos).

```
seq(1, 10, by = 0.5) # secuencia desde 1 a 10 de 0.5 en 0.5

## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0

## [16] 8.5 9.0 9.5 10.0

seq(1, 50, l = 11) # secuencia desde 1 a 100 de longitud 11

## [1] 1.0 5.9 10.8 15.7 20.6 25.5 30.4 35.3 40.2 45.1 50.0
```

Otro atajo que podemos usar para definir secuencias de números con un patrón es definir **vectores repetidos**, por ejemplo un vector lleno de 0, para luego ser llenado (pero ya tenerlo definido). La función `rep()` nos permite repetir un elemento un número fijado de veces.

```
rep(0, 7) # vector de 7 ceros

## [1] 0 0 0 0 0 0 0
```

No solo podemos repetir un número sino que podemos repetir vectores enteros.

```
rep(c(0, 1, 2), 4) # repetimos el vector c(0, 1, 2) 4 veces

## [1] 0 1 2 0 1 2 0 1 2 0 1 2
```

Esa repetición además podemos definirla también de forma **intercalada**: en lugar de repetir `c(0, 1, 2)` cuatro veces seguidas, queremos cuatro 0, después cuatro 1, y después cuatro 2.

```
rep(c(0, 1, 2), each = 4) # cuatro 0, luego cuatro 1, luego cuatro 2

## [1] 0 0 0 0 1 1 1 1 2 2 2 2
```

## 3.2 Operaciones aritméticas con vectores numéricos

Hemos dicho que un número es un vector de longitud 1, así que **toda operación aritmética** que podamos hacer con un número la vamos a poder a hacer con un vector de números, de forma que si hacemos por ejemplo la operación  $2 * z$ , lo que sucederá es **CADA ELEMENTO del vector** será multiplicado 2. De la misma manera se pueden definir sumas  $z + x$ , raíces cuadradas  $\text{sqrt}(z)$  o elevar cada elemento al cuadrado  $z^2$ .

```
z <- c(2, 4, 6)
2 * z

## [1] 4 8 12

x <- 1 + 2
z + x

## [1] 5 7 9

sqrt(z)

## [1] 1.414214 2.000000 2.449490

z^2

## [1] 4 16 36
```

Dado que la operación (por ejemplo, una suma) se realiza elemento a elemento, **¿qué sucederá si sumamos dos vectores de distinta longitud?** Prueba a definir un vector con los 4 primeros impares y súmale a  $z$ .

```

y <- c(1, 3, 5, 7)

variable_suma <- z + y

## Warning in z + y: longitud de objeto mayor no es múltiplo de la longitud de uno

## menor

variable_suma

## [1] 3 7 11 9

```

Como ves, R intenta molestarte lo menos posible, así que lo hace es reciclar: si tiene un vector de 4 elementos y le intentas sumar uno de 3 elementos, lo que hará será **reciclar** elementos del vector con menor longitud: hará  $1+2$ ,  $3+4$ ,  $5+6$  pero...  $7+2$  (vuelve al primer par).

### 3.3 Operaciones estadísticas con vectores numéricos

Al igual que podemos ejecutar operaciones aritméticas, podemos realizar también **operaciones estadísticas** con los vectores, como calcular su suma (`sum()`), su media (`mean()`), su mediana (`median()`), su suma acumulada (`cumsum()`) cada elemento lo acumula al anterior) o percentiles (`quantiles()`).

```

sum(y) # suma

## [1] 16

mean(y) # media

## [1] 4

```

```
median(y) # mediana

## [1] 4

cumsum(y) # suma acumulada

## [1] 1 4 9 16

y <- c(1, 2, 5, 5, 8, 9, 10, 10, 10, 11, 13, 15, 20, 23, 24, 29)

quantile(y, probs = c(0.15, 0.3, 0.7, 0.9)) # Percentiles p15, p30, p70 y p90

## 15% 30% 70% 90%
## 5.0 8.5 14.0 23.5
```

Ver conceptos básicos en [3.10](#).

## 3.4 Vectores de caracteres (texto)

Un error común es asociar vectores solo a números: un **vector** es una colección de elementos del mismo tipo pero no tienen porque ser necesariamente números.

Vamos a crear una frase de ejemplo, un vector de 4 elementos de tipo texto (en R se llaman `char`): "Mi", "nombre", "es" "Javier".

Como ves las variables de tipo `char` van entre comillas dobles, ya que es un **cadena de texto**.

```
mi_nombre <- c("Mi", "nombre", "es", "Javier")

mi_nombre

## [1] "Mi"      "nombre"   "es"      "Javier"
```

Ya tenemos nuestro primer vector de texto de longitud 4. Las **cadenas de texto** son un tipo especial de dato, con los que obviamente no podremos hacer operaciones aritméticas como la suma o la media, pero si podemos hacer operaciones propias de cadenas de texto como puede ser la función `paste()`. Dicha función nos permite convertir un vector de 4 palabras en una frase, decidiendo que carácter queremos que vaya entre palabra con el argumento `collapse =`.

```
paste(mi_nombre, collapse = "") # todo junto
## [1] "MinombreesJavier"

paste(mi_nombre, collapse = " ") # separados por un espacio
## [1] "Mi nombre es Javier"

paste(mi_nombre, collapse = ".") # separados por un punto .
## [1] "Mi.nombre.es.Javier"
```

Si queremos pegar los elementos de la cadena de texto sin ningún tipo de carácter, existe una forma más abreviada y limpia de ejecutar la orden `paste(mi_nombre, collapse = "")`, usando la función `paste0()`

```
paste0(mi_nombre) # todo junto sin nada separando
## [1] "Mi"      "nombre"  "es"      "Javier"
```

Esta función es muy útil si queremos definir variables de texto que comparten por ejemplo un prefijo (`variable_1, variable_2, ...`)

```
paste0("variable", 1:7) # a la palabra «variable» le pegamos los números del 1 al 7

## [1] "variable1" "variable2" "variable3" "variable4" "variable5" "variable6"
## [7] "variable7"

paste("variable", 1:7, sep = "_") # separado por una barra baja

## [1] "variable_1" "variable_2" "variable_3" "variable_4" "variable_5"
## [6] "variable_6" "variable_7"
```

Otra forma más intuitiva de trabajar con textos y variables numéricas es usar el paquete `{glue}`, que nos permite pegar cadenas de texto a variables numéricas de **forma simbólica**.

```
install.packages("glue")
library(glue)
edad <- 10:15 # edades
glue("La edad es de {edad} años")

## La edad es de 10 años
## La edad es de 11 años
## La edad es de 12 años
## La edad es de 13 años
## La edad es de 14 años
## La edad es de 15 años
```

```
# Otra forma sin definir variables a priori
glue("La edad es de {10:15} años")

## La edad es de 10 años
## La edad es de 11 años
## La edad es de 12 años
## La edad es de 13 años
## La edad es de 14 años
## La edad es de 15 años
```

Ya sabemos trabajar con textos :)

¿Y si queremos **pasar todo a mayúscula?** ¿O **todo a minúscula?** ¿Y si queremos **sustituir un carácter (por ejemplo .) por otro en todos los elementos?** R también nos proporciona algunas funciones muy sencillas de usar para dichas tareas. Aquí un ejemplo de algunas de ellas.

```
texto <- c("Hola.", "qué", "ase?", "todo", "bien.", "y yo",
         "que", "ME", "ALEGRO")
toupper(texto) # todo a mayúscula

## [1] "HOLA."  "QUÉ"    "ASE?"   "TODO"   "BIEN."  "Y YO"   "QUE"   "ME"
## [9] "ALEGRO"

tolower(texto) # todo a minúscula

## [1] "hola."  "qué"    "ase?"   "todo"   "bien."  "y yo"   "que"   "me"
## [9] "alegro"
```

```
gsub("o", "*", texto) # toda "o" en el texto será sustituida por *
```

```
## [1] "H*la."  "qué"    "ase?"   "t*d*"   "bien."  "y y*"   "que"    "ME"
```

```
## [9] "ALEGRO"
```



Figure 3.1: Paquete stringr para manejar cadenas de texto más complejas

### 3.5 Vectores lógicos (TRUE/FALSE)

- [X] Variables numéricas (individuales)
  - [X] Vectores de números
  - [X] Vectores de caracteres
  - [ ] Vectores lógicos

Veamos un último tipo de vectores importante en todo lenguaje de programación: los **vectores lógicos**. Un **valor lógico** puede tomar tres valores: TRUE (guardado internamente como un 1), FALSE (guardado internamente como un 0) o NA (dato ausente, son las siglas de *not available*). Estos valores son resultado de evaluar **condiciones lógicas**.

Por ejemplo, imaginemos que definimos un vector de números `x <- c(1.5, -1, 2, 4, 3, -4)`. ¿Qué números del vector son menores que 2? Basta con que ejecutemos la orden `x < 2`, que nos devolverá TRUE/FALSE en cada hueco, en función

de si cumple (**TRUE**) o no (**FALSE**) la condición pedida.

```
x <- c(1.5, -1, 2, 4, 3, -4)
x < 2
## [1] TRUE TRUE FALSE FALSE FALSE TRUE
```

El primer, segundo y sexto elemento del vector son los únicos elementos (estrictamente) menores que 2, de ahí que en el primer, segundo y sexto elemento aparezca un **TRUE** y en el resto un **FALSE**. Es **importante** recordar que al **evaluar una condición lógica sobre un vector** de longitud  $n$ , la **salida sigue siendo un vector** de longitud  $n$  pero con valores lógicos.

Dicha condición lógica puede hacerse con otros operadores como  $\leq$ ,  $>$  o  $\geq$ .

```
x <= 2
## [1] TRUE TRUE TRUE FALSE FALSE TRUE

x > 2
## [1] FALSE FALSE FALSE TRUE TRUE FALSE

x >= 2
## [1] FALSE FALSE TRUE TRUE TRUE FALSE
```

También podemos **comparar si es igual a otro elemento**, para lo que usaremos el operador  $=$ , pudiendo usar también su opuesto  $\neq$  (distinto de).

```
x == 2
## [1] FALSE FALSE TRUE FALSE FALSE FALSE

x != 2
## [1] TRUE TRUE FALSE TRUE TRUE TRUE
```

Las **condiciones pueden ser combinadas**, principalmente de dos maneras:

- **Intersección**: todas las condiciones concatenadas se deben cumplir (conjunction y, operador &) para devolver un TRUE.
- **Unión**: basta con que una de las condiciones concatenadas se cumpla (conjunction o, operador |) para devolver un TRUE.

Por ejemplo, vamos a calcular qué elementos del vector `c(1.5, -1, 2, 4, 3, -4)` sean menores que 3 pero (y) mayores que 0, y los elementos menores que 2 o mayores que 3.

```
x <- c(1.5, -1, 2, 4, 3, -4)
x < 3 & x > 0 # Solo los que cumplen ambas condiciones
```

```
## [1] TRUE FALSE TRUE FALSE FALSE FALSE

x < 2 | x > 3 # Los cumplen al menos una de ellas
```

```
## [1] TRUE TRUE FALSE TRUE FALSE TRUE
```

Como hemos comentado anteriormente, los valores lógicos TRUE/FALSE son guardados internamente como 0/1 por lo que podemos usar **operaciones aritméticas con ellos**. Por ejemplo, si queremos averiguar el número de elementos de un vector que cumplen una condición lógica `< 2`, los que lo hagan tendrán

asignado un 1 y los que no un 0, por lo que basta con sumar el vector lógico para obtener el número de elementos bajo dicha condición.

```
sum(x < 2) # sumamos el vector de TRUE/FALSE --> número de TRUE
```

```
## [1] 3
```

### 3.6 Datos ausentes: NA y NaN

La vida no siempre es perfecta así en muchas ocasiones nos encontraremos con lo que llamamos en estadística un **dato ausente** o *missing value*, un **valor que no tenemos en nuestra variable**, y un ejemplo práctico lo tenemos con los datos de vacunación de covid del Ministerio de Sanidad. Cada día se publicaba un PDF (ya...mal) con los datos de vacunación PERO...no se publican datos los fines de semana: en dichas fechas hay datos que no tenemos, y en R se representan por NA (significa *not available*). Vamos a crear un vector de números con datos ausentes con la orden `x <- c(1, NA, 3, NA, NA, 5, 6)`: el vector tendrá longitud 7 pero en el segundo, cuarto y quinto elemento tendremos un dato faltante, un lugar que no tenemos relleno (pero que no eliminamos).

```
x <- c(1, NA, 3, NA, NA, 5, 6) # Vector numérico con datos faltante
length(x) # longitud del vector
```

```
## [1] 7
```

```
x
```

```
## [1] 1 NA 3 NA NA 5 6
```

¿Puedes aventurar qué sucede cuando multiplicamos ese vector por 2 por ejem-

plo?

```
2 * x # operación aritmética con un vector con NA
```

```
## [1] 2 NA 6 NA NA 10 12
```

Efectivamente: un dato que no tenemos, multiplicado por 2, sigue siendo un dato ausente. Es muy importante para evitar resultados erróneos que entendamos que un **dato ausente no computa en una operación aritmética, es un hueco vacío**. Si hacemos la suma del vector, estamos sumando números más datos ausentes, por lo que el resultado final será también un dato ausente. Si tenemos algún dato ausente en nuestro vector, la suma final está a su vez ausente, ¡no podemos saber cuánto vale!

```
sum(x) # suma de un vector que contiene NA
```

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

Para evitar que un dato ausente en nuestros datos nos impida hacer ciertas operaciones, en muchas funciones de R podemos añadir el argumento `na.rm = TRUE`: **primero elimina los datos ausentes**, y luego ejecuta la función.

```
sum(x, na.rm = TRUE) # eliminando datos ausentes
```

```
## [1] 15
```

Una manera de **localizar que elementos están ausentes** en nuestras variables es con la función `is.na()`, una función que nos devuelve un vector de valores lógico: `TRUE` si el elemento está ausente y `FALSE` si no lo está.

```
is.na(x) # TRUE si está ausente (NA), FALSE si no lo está.
```

```
## [1] FALSE TRUE FALSE TRUE TRUE FALSE FALSE
```

Dichos **datos ausentes se pueden eliminar** (sin necesidad de sumarlos) con la función `na.omit()` (aunque a veces lo que nos interesa es que no sea ausente, introduciendo el punto medio entre su valor anterior y su valor posterior, por ejemplo).

```
na.omit(x)
```

```
## [1] 1 3 5 6
## attr(,"na.action")
## [1] 2 4 5
## attr(,"class")
## [1] "omit"
```

Hay un **tipo de dato muy particular, como resultado de operaciones no permitidas o cuyo resultado es indeterminado**, que en R lo veremos como `NaN`: *not a number*, un resultado fruto de una indeterminación, como por ejemplo la operación `0/0` (cuyo límite no está definido). Importante saber que también existe una forma de denotar al infinito como `Inf`, siendo el resultado de algunas operaciones como `1/0` (cuyo límite si existe).

```
1/0
```

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

```
0/0  
  
## [1] NaN  
  
sqrt(-1)  
  
## Warning in sqrt(-1): Se han producido NaNs  
  
## [1] NaN
```

De la misma manera que podemos localizar valores NA, tenemos a nuestra disposición las funciones `is.infinite()` y `is.nan()` para detectar que elementos de nuestro vector son Inf o NaN, respectivamente.

```
x <- c(1, NA, 3, 4, Inf, 6, 7, Inf, NaN, NA)  
  
is.na(x)  
  
## [1] FALSE TRUE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE  
  
is.nan(x)  
  
## [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE  
  
is.infinite(x)  
  
## [1] FALSE FALSE FALSE FALSE TRUE FALSE FALSE TRUE FALSE FALSE
```

### 3.7 Seleccionar elementos de un vector

Ya sabemos definir variables que sean vectores (recuerda: colección de valores del mismo tipo).

¿Y si del vector original queremos EXTRAER UN SUBCONJUNTO del mismo,

por ejemplo, los primeros 10 elementos?

R tiene varias formas de hacer esto pero la más sencilla es entendiendo que si yo quiero **acceder al elemento i** de un vector, deberé usar el **operador de selección [i]**. Veamos un ejemplo

```
x <- 1:100 # Vector de longitud 100 (del 1 al 100)
y <- x[37] # Solo me interesa el elemento que ocupa el lugar 37
y
```

```
## [1] 37
```

Dado que hemos visto que un número no es más que un vector de longitud uno, esta operación también la podemos aplicar usando un vector de índices a seleccionar, de forma que le podemos indicar simultáneamente que valores que queremos

```
x[c(1, 4, 51, 77)] # Solo queremos acceder a los elementos en la posición 1, 4, 51, y 77
```

```
## [1] 1 4 51 77
```

```
y <- c("hola", "qué", "tal", "todo", "ok", "?")
```

```
y[1:2] # Solo queremos acceder a los elementos en la posición 1 y 2
```

```
## [1] "hola" "qué"
```

```
c(1:2, length(y))
```

```
## [1] 1 2 6
```

```
y[c(1:2, length(y))] # Solo accedemos a los elementos en la posición 1, 2 y además el que ocupa la longitud del vector
```

```
## [1] "hola" "qué"  "?"
```

Otras veces no querremos seleccionar un elemento en concreto sino **filtrar algunos elementos en concreto y no extraerlos**, para lo cual deberemos repetir la misma operación pero con el signo - delante: el operador [-i] no selecciona el elemento i-ésimo del vector sino que lo elimina en nuestro filtro.

```
y
```

```
## [1] "hola" "qué"  "tal"  "todo" "ok"   "?"
```

```
z <- y[-2] # Nos muestra todo y salvo el elemento que ocupa la segunda posición
```

```
z
```

```
## [1] "hola" "tal"  "todo" "ok"   "?"
```

Sin embargo, lo habitual es que dicho filtro que hagamos de una variable lo **hagamos en base a una condición lógica**. Supongamos que x <- c(7, 20, 18, 3, 19, 9, 13, 3, 45) y y <- c(17, 21, 58, 33, 15, 59, 13, 1, 45) son las edades de dos grupos de personas y que queremos quedarnos solo con los mayores edad. ¿Tenemos que andar averiguando en qué posición se encuentran para luego seleccionarlos? No, vamos a **seleccionar los elementos que cumplen una condición dada**.

```
x <- c(7, 20, 18, 3, 19, 9, 13, 3, 45)
```

```
y <- c(17, 21, 58, 33, 15, 59, 13, 1, 45)
```

```
x[x >= 18] # mayores de 18 años del conjunto x
```

```
## [1] 20 18 19 45
```

```
y[x >= 18] # mayores de 18 años del conjunto y
```

```
## [1] 21 58 15 45
```

Lo que hemos hecho ha sido pasarlo como índices un vector lógico TRUE/FALSE, de forma que solo filtrará los que tengan un TRUE asignado, aquellos que cumplen la condición lógica introducida. Esto también nos puede servir para limpiar de datos ausentes, combinando la función `is.na()`, que nos localiza el lugar que ocupan los ausentes, con el operador `!`, que lo que hace es negar lo que venga detrás. También podemos probar a **combinar condiciones lógicas para nuestra selección**.

```
x <- c(7, NA, 20, 3, 19, 21, 25, 80, NA)
```

```
x[x >= 18] # mayores de 18 años del conjunto x
```

```
## [1] NA 20 19 21 25 80 NA
```

```
x[is.na(x)] # solo valores ausentes
```

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

```
x[!is.na(x)] # sin valores ausentes: ! es el símbolo de la negación
```

```
## [1] 7 20 3 19 21 25 80
```

```
!(x >= 18) # niega los mayores de 18 años, todo lo que no cumpla esa condición
```

```
## [1] TRUE     NA FALSE   TRUE FALSE FALSE FALSE FALSE    NA
```

```
x[x >= 18 & x <= 25] # los valores que cumplen ambas (&): entre 18 y 25 años
```

```
## [1] NA 20 19 21 25 NA
```

Como ves si un valor es NA, la evaluación de una condición lógica sobre él (mayor o menor de 18 años) nos seguirá devolviendo NA. Por último, R nos permite **dar significado léxico** a nuestros valores (significan algo, no solo números), pudiendo poner nombres a los elementos de un vector, permitiendo su selección por dichos nombres

```
x <- c("edad" = 31, "tlf" = 613910687, "cp" = 33007) # cada número tiene un significado distinto
x
```

```
##      edad        tlf        cp
```

```
##      31 613910687      33007
```

```
x[c("edad", "cp")] # seleccionamos los elementos que tienen ese nombre asignado
```

```
##  edad    cp
```

```
##  31 33007
```

Con la función `names()` además podemos, no solo consultar los nombres de una variable, sino cambiarlos a nuestro gusto.

```
names(x) # Consultamos nombres
```

```
## [1] "edad" "tlf"  "cp"
```

```
names(x) <- c("años", "móvil", "dirección") # Cambiamos nombres
```

```
names(x) # Consultamos nuevos nombres
```

```
## [1] "años"     "móvil"     "dirección"
```

```
x
##      años     móvil dirección
##      31 613910687      33007
```

### 3.7.1 which

Hemos visto como seleccionar elementos de un vector que cumplen una condición, para a veces no queremos el elemento en sí, sino el lugar que ocupa: **¿qué valores de un vector cumplen una condición lógica, qué lugar ocupan?** Para obtener dicho índice tenemos a nuestro disposición la función `which()`, que no nos devuelve el elemento en sí sino su lugar.

```
x <- c(7, NA, 20, 3, 19, 21, 25, 80, NA)
x[x >= 18] # Accedemos a los elementos que cumplen la condición
## [1] NA 20 19 21 25 80 NA

which(x >= 18) # Obtenemos los lugares que ocupan los elementos que cumplen la condición
## [1] 3 5 6 7 8
```

Esta función es muy útil especialmente cuando queremos **averiguar el valor que ocupa el máximo/mínimo** de una colección de valores, con las funciones `which.max()` y `which.min()`.

```
max(x, na.rm = TRUE) # máximo de x (si no eliminamos NA, nos devolverá NA)
## [1] 80
```

```
min(x, na.rm = TRUE) # mínimo de x (si no eliminamos NA, nos devolverá NA)

## [1] 3

which.max(x) # Lugar que ocupa el máximo

## [1] 8

x[which.max(x)]


## [1] 80

which.min(x) # Lugar que ocupa el mínimo

## [1] 4

x[which.min(x)]


## [1] 3
```

### 3.7.2 NULL

A veces veremos que además de NA y NaN, R nos muestra un dato llamado NULL. Cuando tenemos NA en alguna variable, el registro existe, pero no está lleno. Sin embargo, cuando tenemos un NULL significa que ese registro ni siquiera existe: no es un dato guardado pero cuyo valor desconocemos, es un dato que ni siquiera existe (por ejemplo, si guardamos datos de 7 personas, el dato de la octava persona no es NA, es que no hay octava persona directamente).

```
x <- c(1, NA, 3, NA, NA, 5, 6)

x[2] # NA: el registro existe pero sin dato
```

```
## [1] NA

names(x) # No hemos definido el nombre de las variables, así que devuelve NULL

## NULL
```

### 3.8 Ordenar vectores

Una acción habitual al trabajar con datos es **saber ordenarlos**: de menor a mayor edad, datos más recientes vs antiguos, etc. Para ello tenemos la función `sort()`, que podemos usar directamente para **ordenar de menor a mayor**, o con el argumento `decreasing = TRUE`, para **ordenar de mayor a menor**.

```
x <- c(1, -3, 0, 10, 5, 2, 7, -13)

sort(x) # orden de menor a mayor

## [1] -13 -3 0 1 2 5 7 10

sort(x, decreasing = FALSE) # orden de mayor a menor

## [1] -13 -3 0 1 2 5 7 10
```

Otra forma de ordenar un vector es que R nos **devuelva los índices de los elementos ordenados**, y luego usar dichos índices para reorganizar los elementos, con la función `order()`.

```
order(x) # el elemento más pequeño es el octavo, luego el segundo, luego el tercero, lue

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

```
x[order(x)] # accedemos a los índices ordenados, equivalente al sort(x)

## [1] -13   -3    0    1    2    5    7   10
```

## 3.9 Fechas

Hay un tipo muy especial de datos que son los **datos tipo fecha**. Una fecha podría ser a priori una simple cadena de texto "2021-04-21" pero podemos usar la función `as.Date()` para que R entienda que esa cadena de texto representa un instante temporal. Fíjate la diferencia entre una fecha en texto y una fecha con `as.Date()`.

```
fecha_char <- "2021-04-21"

fecha_date <- as.Date(fecha_char, format = "%Y-%m-%d")

fecha_char + 1

## Error in fecha_char + 1: argumento no-numérico para operador binario

fecha_date + 1

## [1] "2021-04-22"
```

En el momento en que el convertimos la cadena de texto a fecha, aunque se visualice como tal, internamente es un número, por lo que podemos restar fechas (días entre ambas), **podemos sumar números a fechas (fecha días después)**, etc.

Dentro del entorno `{tidyverse}`, el paquete `{lubridate}` tiene implementadas **múltiples funciones para poder operar con fechas de forma sencilla e intuitiva**.

### 3.10 □ Glosario

- **numeric:** variables de tipo numéricas (algunas veces vendrán indicados como `int` o `integer` para enteros, y `dbl` o `double` para números con decimales).
- **character:** variables de tipo carácter.
- **Date:** variables de tipo fecha.
- **Media:** medida de centralización que consiste en sumar todos los elementos y dividirlos entre la cantidad de elementos sumados. A pesar de ser la más conocida, la media es **muy poco robusta**: dado un conjunto, si se introducen **valores atípicos o outliers** (valores muy grandes o muy pequeños), la media se perturbará con mucha facilidad. Dado un vector de valores  $x = (x_1, \dots, x_n)$ , se denota como  $\bar{x}$ .

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

- **Mediana:** medida de centralización que consiste en, tras ordenar los datos de menor a mayor, quedarnos con el valor que ocupa el medio (deja tantos números por debajo como por encima). Más robusta que la media aunque menos la moda. Dado un vector de valores  $x = (x_1, \dots, x_n)$ , se denota como  $Me_x$ .

$$Me_x = \arg \min_{x_i} \{F_i > 0.5\}, \quad F_i = \frac{\#\{x_j \leq x_i\}}{n}$$

- **Moda:** medida de centralización que consiste en encontrar el valor más repetido (el valor *trending*). Es la medida de centralización más robusta. Dado un vector de valores  $x = (x_1, \dots, x_n)$ , se denota como  $Mo_x$ .

$$Mo_x = \arg \max_{x_i} f_i, \quad f_i = \frac{\#\{x_j = x_i\}}{n}$$

## 3.11 Consejos y tips

Operaciones elemento a elemento (vectorial)

Es importante recordar que cada operación con un vector es una **operación realizada en CADA elemento del vector, devolviéndonos a su vez un vector** de salida de igual longitud que la variable con la que hemos operado.

Diferencia de conjuntos

Una función muy útil para ver las **diferencias entre dos conjuntos** es `setdiff()`, una función que nos devuelve los elementos distintos entre dos conjuntos.

```
y <- 1:10
z <- c(1, 3, 7, 10)
setdiff(y, z) # Elementos en y que no están en z
```

```
## [1] 2 4 5 6 8 9
```

Argumentos por defecto

La función `sort()` es un buen ejemplo de que las **funciones traen definidos argumentos por defecto** (aunque no los veamos a priori). La orden `sort(x)` en realidad está ejecutando `sort(x, decreasing = TRUE)`, pero como es su valor por defecto, nos podemos ahorrar incluirlo. Escribe `? help sort()` en la consola y verás como en la cabecera de la función ya hay preasignado un `decreasing = TRUE`.

### Recuperar un comando y autocompletar

Si haces click con el ratón en la consola y pulsas la flecha «arriba» del teclado, te irá apareciendo todo el **historial de órdenes ejecutadas**. Es una manera de ahorrar tiempo para ejecutar órdenes similares.

Si empiezas a escribir el nombre de una variable pero no te acuerdas exactamente de su nombre, si pulsas **tabulador**, R te **autocompletará** solo (prueba a escribir solo `variab` y pulsa tabulador)

### all, any

Existen dos funciones muy útiles en R para saber si **TODOS** o **ALGUNO** de los elementos de un vector cumple una condición. Las funciones `all()` y `any()` nos devolverá un único valor lógico. Estas funciones son muy útiles al final de los códigos para comprobar que las condiciones que tienen que verificar los datos se cumplen, y asegurarnos que el proceso se ha ejecutado correctamente (por ejemplo, que todos los datos sean positivos o no haya datos ausentes).

```
x <- c(1, 2, 3, 4, 5, NA, 7)
```

```
all(x < 3)
```

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

```
any(x < 3)
```

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

```
all(x > 0)
```

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

```
all(is.na(x) > 0)
```

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

```
all(is.na(x))
```

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

```
any(is.na(x))
```

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

Constantes: número pi

R tiene una variable reservada al número  $\pi$ , lista para ser usada, por lo que se recomienda no nombrar a ninguna variable con dicho nombre.

```
pi
```

```
## [1] 3.141593
```

### Convertir tipos de datos

A veces la lectura de variables numéricas de nuestros archivos puede hacer que un número, por ejemplo 1, sea leído como la cadena de texto "1", con la que no podemos operar como un número. Las funciones `as.numeric()`, `as.character()` y `as.logical()` nos permiten convertir una variable en tipo numérico, carácter o lógico, respectivamente.

```
"1" + 1
## Error in "1" + 1: argumento no-numérico para operador binario

as.numeric("1") + 1
## [1] 2

as.character(1)
## [1] "1"

as.logical(c(0, 1))
## [1] FALSE TRUE
```

### Optimizar nuestro código: eficiencia en tiempo de ejecución

Aunque parezca un tema menor, si tu código tarda 1 milisegundo más de lo que podría tardar de otra forma, si esa orden se repite muchas veces, ese milisegundo extra puede ser 5, 10 o 20 minutos más que tu código tardará en ejecutarse. Hay

un paquete muy útil en R para medir tiempos de distintas órdenes que hacen lo mismo (el paquete `{microbenchmark}`), vamos a instalarlo.

```
install.packages("microbenchmark")
library(microbenchmark)
```

Este paquete contiene una orden para comparar el tiempo de dos órdenes: necesita como primeros argumentos las dos órdenes cuyos tiempos vamos a comparar, y un argumento `times` en el que le indicamos el número de veces que ejecutará cada orden para realizar los tiempos medios. Vamos a comparar los comandos de ordenación `order()` y `sort()`.

```
x <- rnorm(1e3) # 1000 elementos aleatorios de una normal N(0, 1)
microbenchmark(sort(x), # primera forma
               x[order(x)], # segunda forma
               times = 1e3) # se repetirá 1000 veces

## Unit: microseconds
##      expr     min      lq      mean    median      uq      max neval cld
##      sort(x) 125.061 240.5335 650.4693 261.7675 309.6525 169912.76 1000  a
## x[order(x)]  91.097 167.0730 359.4541 180.8995 209.7690  53435.15 1000  a
```

Sí, estás viendo bien: aunque a priori parezca contraintuitivo, es más corto obtener los índices ordenados de un vector, y luego reordenarlo en base a esos índices, que la ordenación directa a través del comando `sort()` (ya que usan algoritmos de ordenación distintos).

### 3.12 □ Ejercicios

□ Ejercicio 1: define un vector que contenga los números 1, 10, -1 y 2, y guárdalo en una variable llamada `vector_num`. Tras definirlo, calcula su suma y la versión ordenada del vector definido como sumar 1 a cada elemento de `vector_num`.

- Solución:

```
# Vector de números

vector_num <- c(1, 10, -1, 2)

# Suma

sum(vector_num)

## [1] 12

# Ordenamos el vector + 1 (con sort)

sort(vector_num + 1)

## [1]  0  2  3 11

# Ordenamos el vector + 1 (con order)

vector_num2 <- vector_num + 1

vector_num2[order(vector_num2)]

## [1]  0  2  3 11
```

□ Ejercicio 2: encuentra del vector `vector_num` original el lugar (el índice) que ocupa su mínimo y su máximo. Devuelve un vector lógico con los elementos que son mayores 1 y menores que 7. Piensa una manera de encontrar si todos son

positivos.

- Solución:

```
vector_num <- c(1, 10, -1, 2)

# Encontrando el lugar que ocupa el máximo y mínimo

which.max(vector_num)
```

```
## [1] 2
```

```
which.min(vector_num)
```

```
## [1] 3
```

```
# Vector lógico: mayores que 1 y menores que 7

vector_num > 1 & vector_num < 7
```

```
## [1] FALSE FALSE FALSE  TRUE
```

```
# ¿Son todos positivos?

all(vector_num > 0)
```

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

□ Ejercicio 3: crea un vector con las palabras “Hola”, “me”, “llamo” (y tu nombre y apellidos), y pega luego sus elementos de forma que la frase esté correctamente escrita en castellano. Tras hacerlo, añade “y tengo 30 años”.

- Solución:

```
# Definiendo el vector

vector_char <- c("Hola", "me", "llamo", "Javier",
               "Álvarez", "Liébana")

# Pegamos

paste(vector_char, collapse = " ")

## [1] "Hola me llamo Javier Álvarez Liébana"

# Añadimos frase

paste0(paste(vector_char, collapse = " "), " y tengo 30 años.")

## [1] "Hola me llamo Javier Álvarez Liébana y tengo 30 años."
```

- Ejercicio 4: obtén la fecha de hoy, define la fecha de tu cumpleaños, y calcula la diferencia de días.

- Solución:

```
# Hoy

hoy <- Sys.Date()

# Cumple (diferentes formatos)

cumple <- as.Date("1989-09-10")

cumple <- as.Date("10-09-1989", "%d-%m-%Y")

# Diferencia

hoy - cumple
```

```
## Time difference of 11686 days
```



## Chapter 4

# Flujo de trabajo: proyecto

Estamos listos/as para crear nuestro **primer proyecto de R**:

Cuando se empieza a programar para un trabajo concreto de R es recomendable crearnos lo que se conoce como un **proyecto de R**: en lugar de ir abriendo ventanas sueltas para programar (los scripts, los archivos con extensión .R), podemos **agruparlos** en distintos proyectos, de forma que podamos acceder a ellos de forma ordenada (algo así como crear carpetas en nuestro disco duro).

### 4.1 Crear proyecto

Para crear nuestro proyecto deberemos de ir al menú superior `File << New Project` (ver [4.1](#))

Se nos abrirá una ventana con 3 opciones:

- **New directory**: crear un proyecto desde el inicio (**opción recomendable**).

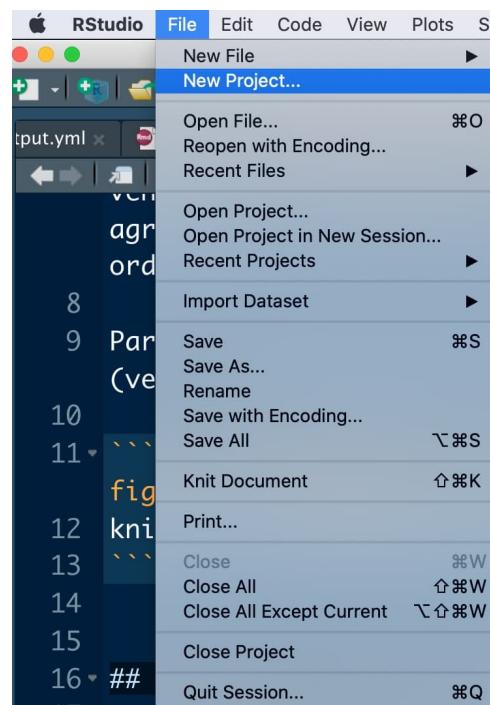


Figure 4.1: Crear un nuevo proyecto en R.

- **Existing directory:** crear un proyecto con los códigos que tienes ya guardados en una carpeta.
- **Version control:** para importar el proyecto de algún repositorio y vincularlo a él.

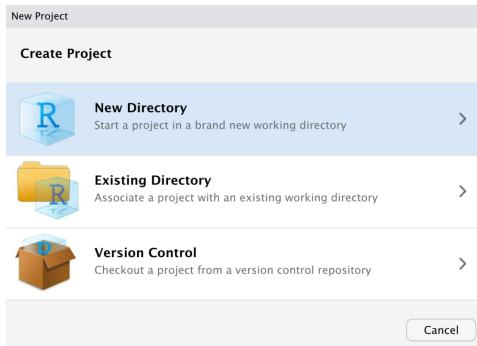


Figure 4.2: Opciones de creación.

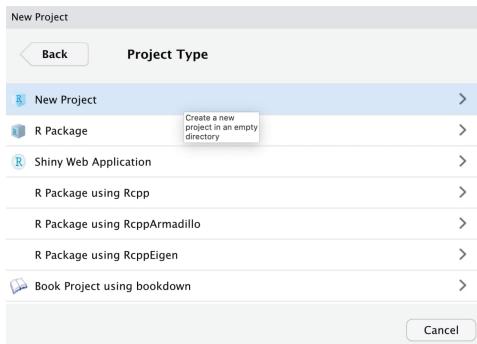


Figure 4.3: Clickar en «New project».

Deberemos elegir el **directorio de nuestro ordenador** donde queremos que se guarde (una carpeta que contendrá todos los códigos y datos de ese proyecto), así como el **nombre del proyecto** (que será a su vez el nombre de la subcarpeta que se os creará en el ordenador).

Una vez que el proyecto está creado, abriremos nuestro primer **script de R** (donde

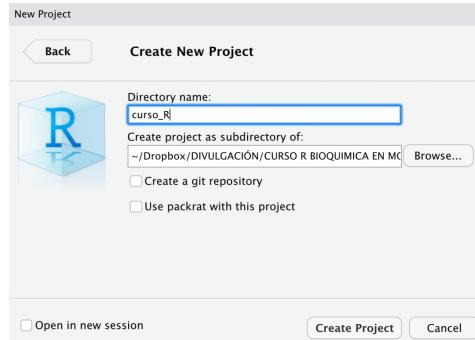


Figure 4.4: Nombre del proyecto.

escribiremos el código), escribiremos una descripción del proyecto en la primera línea y guardaremos el archivo (archivo de extensión .R).

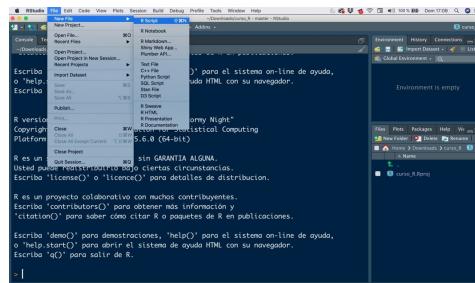


Figure 4.5: Abrir nuestro primer script de R.

Este será nuestro **código principal** (puedes ponerle el nombre que quieras, normalmente se le llama `main.R` para diferenciarlo del resto), desde el que iremos construyendo nuestro código e iremos llamando a otros archivos si es necesario. Recuerda que **programar es como escribir**: cuanto más limpio y estructurado, mejor se entenderá.

**La ventaja de tener los códigos agrupados por proyectos** es que si estamos tra-

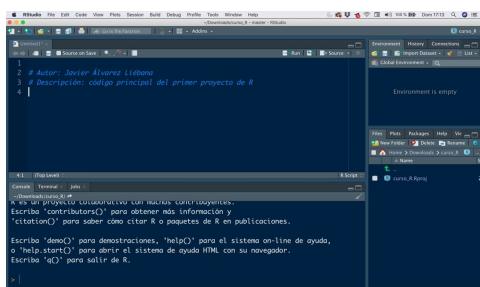


Figure 4.6: Descripción al inicio del código.

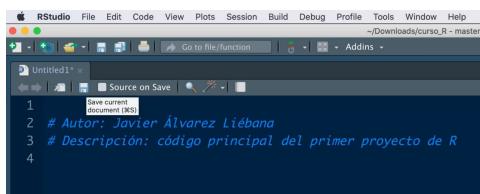


Figure 4.7: Guardamos el código.

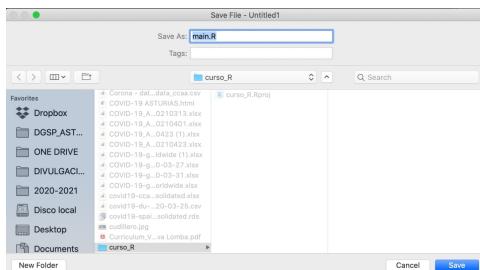


Figure 4.8: Guardamos el código.

jando en varios a la vez podemos saltar de uno a otro, visualizando solo los códigos de un proyecto, y no los 100 archivos `sin_titulo131.R` que vayamos creando.

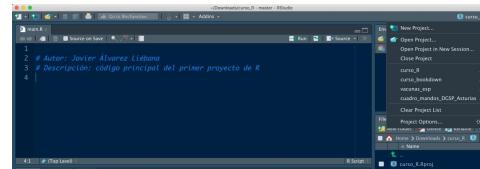


Figure 4.9: Saltar de proyecto en proyecto.

## 4.2 Directorios de trabajo y cabecera

Como luego veremos en la 5, es altamente recomendable que **todos los archivos (códigos, datos, imágenes, recursos, etc)** los tengamos dentro de la misma carpeta del proyecto (aunque podamos crear subdirectorios), para que trabajar en el proyecto sea más sencillo e intuitivo. Vamos a crearnos dentro de la carpeta del proyecto, una subcarpeta que se llame **CODIGOS**, y creamos un script llamado **variables.R** dentro de esa carpeta.

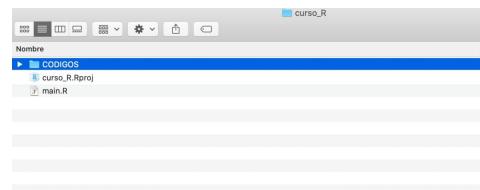


Figure 4.10: Subcarpeta «CODIGOS».

En ese código de prueba vamos a **definir algunas variables fijas** que luego usaremos en el código principal (suele suceder con variables que van a ser fijas como nombres, fechas o codificaciones de variables).

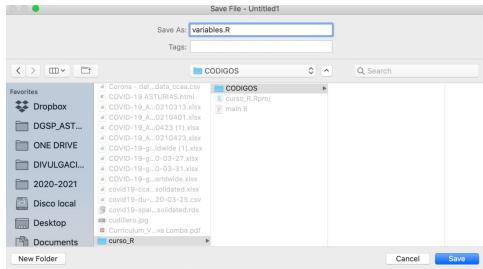


Figure 4.11: Creamos un fichero «variables.R».

```
# Descripción: script de prueba con variables

# Variables
x <- c(1, 2, 0, -1, 71) # Vector de números
y <- c("hola", "me", "llamo", "Javier") # Vector de caracteres
apellido <- "Álvarez"

# Fechas
hoy <- as.Date(Sys.time()) # Convertir a tipo fecha la fecha de hoy
fecha_origen <- as.Date("2021-01-01") # Inicio de año
```

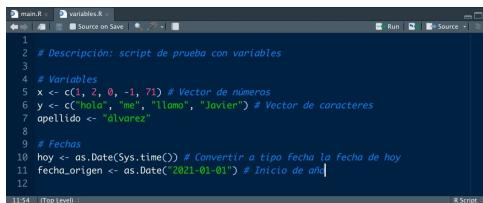


Figure 4.12: Escribimos una serie de variables fijas para luego ser usadas.

No es obligatorio pero es **altamente recomendable** tener muy estructurado nuestros códigos, de forma que el archivo .R haga una **tarea concreta y definida** (uno

carga archivos, otro preprocesa, otro hace un modelo, otro hace las gráficas), y sea el **código principal el que haga una llamada limpia a todos ellos**, para que en caso de error, la detección del mismo sea más sencilla.

Así que eso haremos: desde nuestro archivo principal `main.R` llamaremos a ese archivo `variables.R`, para luego usar las variables definidas en él.

### ¿Cómo indicarle a R donde está nuestro fichero?

En R, como en todo lenguaje de programación, podemos consultar lo que el ordenador llama *directorio de trabajo*: la carpeta «base» desde donde está ejecutando tu código. Dicha ruta de directorio se puede consultar con la función `getwd()`, pudiendo ver los archivos y carpetas que hay dentro del mismo con el comando `dir()`

```
getwd()
dir()
```

```
Console Terminal Markdown Jobs
~/Dropbox/DIVULGACIÓN/CURSO R BIOQUÍMICA EN MOVIMIENTO/curso_bookdown/ >
> getwd()
[1] "/Users/javieralvarezliebana/Dropbox/DIVULGACIÓN/CURSO R BIOQUÍMICA EN MOVIMIENTO/cu
rso_bookdown"
> dir()
[1] ".book"           "_bookdown_files"   ".bookdown.yml"
[4] ".output.yml"    "02-toma_contacto.Rnd" "03-primer_proyecto.Rmd"
[7] "04-carga_datos.Rmd" "20-biblio.Rmd" "book.bib"
[10] "curso_bookdown.pdf" "curso_bookdown.Rproj" "curso_bookdown.tex"
[13] "imagenes"        "index.Rmd"       "packages.bib"
[16] "preamble.tex"    "README.html"    "README.md"
[19] "requisitos.html" "style.css"      "testdir"
```

Figure 4.13: Consultar directorio de trabajo predeterminado y archivos contenidos en él.

Lo ideal es empezar el código **fijando como directorio de trabajo el directorio donde tengamos nuestro archivo principal `main.R`** y para ello usaremos la función `setwd()`, cuyo argumento será la ruta donde queremos fijarlo. Para hacerlo de forma automática (y que el código pueda ser abierto por ti pero

también por otros que no tengan tu misma estructura de carpetas), obtendremos de forma automática la ruta del archivo `main.R` o del proyecto con la orden `rstudioapi::getSourceEditorContext()$path`, y después usaremos `dirname()` para quedarnos solo con la ruta de carpetas (eliminando el nombre del fichero al final). Esa será la ruta que le pasaremos a `setwd()`, quedando nuestro directorio de trabajo automáticamente fijado, sin preocuparnos de la ruta

```
# Fijamos directorio de trabajo automáticamente
setwd(dirname(rstudioapi::getSourceEditorContext()$path))
```

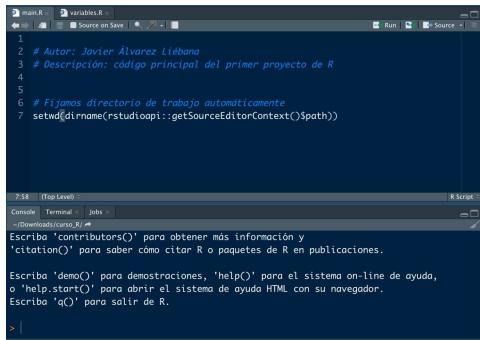
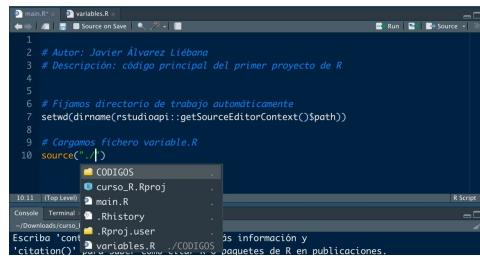


Figure 4.14: Fijamos de forma automática el directorio de trabajo.

Una vez que hemos fijado nuestro directorio, para cargar código `.R`, basta que usemos la función `source()`, cuyo argumento será la ruta del archivo. Como tenemos de directorio base el directorio en el que tenemos nuestro archivo principal (`./`), bastará que empecemos a escribir `source("./")`, presionar el tabulador, y se nos abrirá el menú de archivos de nuestro directorio de trabajo, pudiendo ir seleccionando de forma sencilla la ruta de nuestro archivo.

```
# Cargamos fichero variable.R
source("./CODIGOS/variables.R")
```

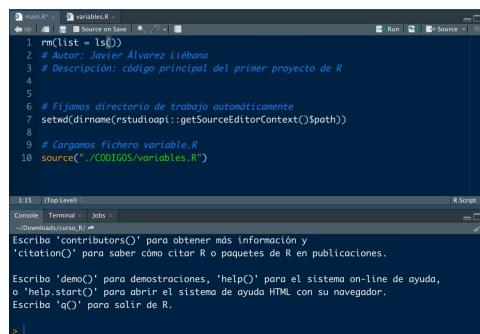


```

1 # Autor: Javier Álvarez Liébana
2 # Descripción: código principal del primer proyecto de R
3
4
5
6 # Fijamos directorio de trabajo automáticamente
7 setwd(dirname(rstudioapi::getSourceEditorContext()$path))
8
9 # Cargamos fichero variable.R
10 source("./")
    CODIGOS
      curso.Rproj
11 main.R
  .Rhistory
  .Rprofile.user
  variables.R
  ./CODIGOS/as información y
  'citation()' para saber cómo citar R o paquetes de R en publicaciones.

```

Figure 4.15: Cargar archivos de nuestro directorio de trabajo.



```

1 rm(list = ls())
2 # Autor: Javier Álvarez Liébana
3 # Descripción: código principal del primer proyecto de R
4
5
6 # Fijamos directorio de trabajo automáticamente
7 setwd(dirname(rstudioapi::getSourceEditorContext()$path))
8
9 # Cargamos fichero variable.R
10 source("./CODIGOS/variables.R")

11  [Top Level] :
  Console  Terminal  Jobs
  . /Downloads/curs.R ↗
Escriba 'contributors()' para obtener más información y
'citation()' para saber cómo citar R o paquetes de R en publicaciones.

Escriba 'demo()' para demostraciones, 'help()' para el sistema on-line de ayuda,
o 'help.start()' para abrir el sistema de ayuda HTML con su navegador.
Escriba 'q()' para salir de R.
> 

```

Figure 4.16: Cargar archivos de nuestro directorio de trabajo.

### 4.3 Ejecución

Ese archivo que hemos incluido en el código principal nos **cargará las variables que hemos definido en él**, pudiendo usarlas en el código. Vamos a ejecutar lo que tenemos de momento, y para ello tenemos 2 opciones: o copiar el código del script en la consola y pulsar *ENTER*, o bien, activando la casilla *source on save* y guardando el script (no solo se guardará sino que se ejecutará).

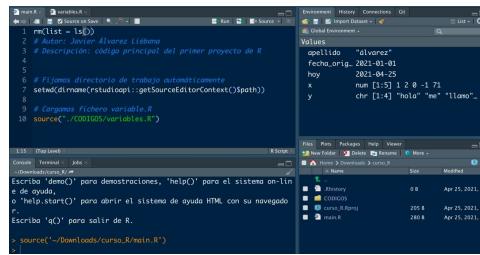


Figure 4.17: Guardamos con la casilla «source on save» activada para que además de guardar se ejecute el código.

Como vemos en la imagen 4.17, una vez ejecutado, tenemos en nuestro panel de entorno (parte superior derecha) las variables ya cargadas que teníamos definidas en nuestro fichero `variables.R`.

Prueba a escribir algunas funciones que hemos aprendido con dichas variables y vuelve a hacer click en «guardar» con *source on save* activado.

```
# Sumamos 3 a cada elemento de x
z <- x + 3
z
```

```
## [1] 4 5 3 2 74
```

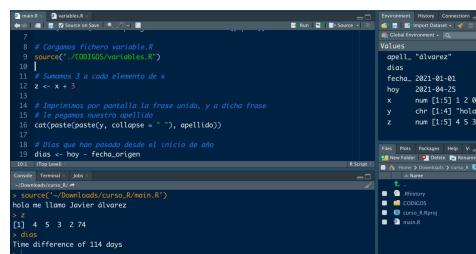
```
# Imprimimos por pantalla la frase unida, y a dicha frase
# le pegamos nuestro apellido
cat(paste(paste(y, collapse = " "), apellido))
```

```
## hola me llamo Javier Álvarez
```

```
# Días que han pasado desde el inicio de año
dias <- hoy - fecha_origen
dias
```

```
## Time difference of 250 days
```

La función `cat()` nos muestra por consola el texto que le pasemos de argumento  
**(función especial para mensajes de alerta por consola)**



```
7
8 # Corremos el fichero variables.R
9 source("./C001005/variables.R")
10 
11 # Sumamos 3 a cada elemento de x
12 # z <- x + 3
13 
14 # Imprimimos por pantalla la frase unida, y a dicha frase
15 # le pegamos nuestro apellido
16 cat(paste(paste(y, collapse = " "), apellido))
17 
18 # Días que han pasado desde el inicio de año
19 dias <- hoy - fecha_origen
20 
21 # Imprimiendo los resultados
22 print(apellido)
23 print(dias)
24 print(Fecha)
25 print(hoy)
26 print(x)
27 print(y)
28 print(z)

[1] "alvarez"
[1] 250
[1] "2021-01-01"
[1] "2021-04-25"
[1] num [1:5] 1 2 0.
[1] chr [1:4] "hola."
[1] num [1:5] 4 5 3.
```

Figure 4.18: Cálculos con las variables definidas: suma, concatenación de texto y diferencia de fechas.

Ya hemos ejecutado nuestro primer proyecto en .R :)

## 4.4 Consejos y tips

Comentarios en los códigos

Es crucial que intentes **documentar al máximo tu código** y que te acostumbres

a ello desde el principio, dejando explícito que haces en cada paso, tanto para ti como para otra persona que pueda leer tu código y lo entienda. Para ello usaremos `# comentario` cuando queramos dejar comentarios en el código. Dichas partes, amén de estar en otro color, no son leídas por R ni ejecutadas: son comentarios que el programa «no ve», solo son para nosotros.

#### Limpiar el entorno

A veces empezamos a programar sin apagar el ordenador y tenemos variables guardadas de otros días que pueden generar conflictos y consumo de memoria. Para asegurarnos que cada vez que empezamos, lo hacemos de cero, es altamente recomendable empezar el código con `rm(list = ls())`. La función `ls()` nos devuelve todas las variables que tenemos definidas en nuestro entorno, y la función `rm()` nos las elimina.

#### Anular warnings

Algunas funciones pueden arrojarse ciertas advertencias que nunca está de más leer. Pero si dichos mensajes de alerta los tenemos controlados y no queremos que nos ensucie la ejecución en la consola, podemos poner al inicio del código `assign("last.warning", NULL, envir = baseenv())` para limpiar los warnings antiguos y `options(warn = -1)` para desactivarlos.

#### Limpiar consola

Podemos limpiar la consola clickando en al escoba que tenemos en la parte superior derecha de la misma. Esta acción **no nos elimina ninguna variable**, simplemente nos limpia la consola de mensajes.

### Guardar los scripts

Los scripts que tengas sin guardar tendrán un asterisco \* al final del nombre en la pestaña superior de la ventana.

### Fecha y hora de hoy

La función `Sys.time()` accede al sistema de nuestro ordenador para decírnos la **fecha y hora del momento de la ejecución** de dicha función.

#### `Sys.time()`

```
## [1] "2021-09-08 12:50:00 CEST"
```

### Cambiar la notación exponencial

Por defecto, R muestra los números en formato de notación exponencial. Por ejemplo, el número 1000000 nos lo mostrará por defecto como `1e+06`. A veces podemos querer que se muestre con todas sus cifras (por ejemplo, en el título o leyenda de una gráfica): para anular la notación exponencial, escribe al inicio del código `options("scipen" = 10)`.

# Chapter 5

## Tipos de datos II: tablas

Sabemos un poco de la gramática y ortografía de nuestro lenguaje, y sabemos las funcionalidades básicas de nuestro Word. Vamos a encontrar la mejor trama para la novela: **hablemos de estructuras de datos**.

### 5.1 Matrices

Hasta ahora **hemos visto solo datos en una dimensión**: una variable, que tiene n valores numéricos, n valores lógicos o n valores de tipo texto. **Una sola variable** (de n elementos).

Pero cuando analizamos datos solemos tener varias variables distintas. Cuando tenemos **distintas variables numéricas de igual longitud**, un formato de dato muy habitual de trabajar es lo que conocemos como **matrices**: una «tabla» de números, con filas y columnas.

Vamos a definir las **edades, teléfonos y códigos postales de una serie de individuos**.

```
edades <- c(14, 24, 56, 31, 20, 87, 73) # vector numérico de longitud 7
tlf <- c(NA, 683839390, 621539732, 618211286, NA, 914727164, NA)
cp <- c(33007, 28019, 37005, 18003, 33091, 25073, 17140)
```

Hasta ahora, cada variable la hemos definido por separado, pero ahora vamos a juntarlas: vamos a crear nuestro **primer conjunto de datos** juntando todas ellas en una matriz, un conjunto de números organizado en **3 columnas (una por variable)** y **7 filas o registros (una por persona)**. Para ello usaremos la función `cbind()`, que nos concatena vectores de igual longitud en formato columna.

```
x <- cbind(edades, tlf, cp) # Construimos la matriz por columnas
```

```
x
```

```
##      edades      tlf      cp
## [1,]     14      NA 33007
## [2,]     24 683839390 28019
## [3,]     56 621539732 37005
## [4,]     31 618211286 18003
## [5,]     20      NA 33091
## [6,]     87 914727164 25073
## [7,]     73      NA 17140
```

Lo que tenemos es una columna por variable y una fila por registro. También podemos construir la matriz por filas con el comando `rbind()` (aunque lo habitual es tener cada variable en una columna).

```
y <- rbind(edades, tlf, cp) # Construimos la matriz por filas  
y  
  
## [,1] [,2] [,3] [,4] [,5] [,6] [,7]  
## edades 14 24 56 31 20 87 73  
## tlf NA 683839390 621539732 618211286 NA 914727164 NA  
## cp 33007 28019 37005 18003 33091 25073 17140
```

Como ves, ahora nuestros datos están **tabulados**, tienen dos dimensiones. ¿Cómo saber las **dimensiones** que tiene una matriz? Prueba a ejecutar la función `dim()`.

```
dim(x)
```

```
## [1] 7 3
```

```
dim(y)
```

```
## [1] 3 7
```

Fíjate que `dim()` devuelve un vector de 2 elementos, por lo que para acceder las filas deberemos ejecutar `dim(x)[1]` (y `dim(x)[2]` para las columnas). También tenemos a nuestra disposición las funciones `nrow()` y `ncol()`, que nos devuelven directamente el número de filas y columnas.

```
dim(x)[1]
```

```
## [1] 7
```

```
dim(x)[2]
```

```
## [1] 3
```

```
nrow(x)
```

```
## [1] 7
```

```
ncol(x)
```

```
## [1] 3
```

Bien, ya sabemos definir una matriz a partir de variables. Igual que a veces es útil generar un vector de elementos repetidos, también podemos definir una **matriz de números repetidos** (por ejemplo, de ceros), con la función `matrix()`, indicándole el número de filas y columnas.

```
matrix(0, nrow = 5, ncol = 3) # 5 filas, 3 columnas, todo 0's
```

```
##      [,1] [,2] [,3]
## [1,]     0     0     0
## [2,]     0     0     0
## [3,]     0     0     0
## [4,]     0     0     0
## [5,]     0     0     0
```

También podemos definir una **matriz a partir de un vector numérico**, reorganizando los valores en forma de matriz (con una dimensión tal que `filas * columnas = longitud del vector`), sabiendo que los elementos se van colocando por columnas (primeros valores en la primera columna, de arriba a abajo).

```
z <- matrix(1:15, ncol = 5) # Matriz con el vector 1:5 con 5 columnas (ergo 3 filas)
z
```

```

##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    4    7   10   13
## [2,]    2    5    8   11   14
## [3,]    3    6    9   12   15

class(z) # Clase de la variable

## [1] "matrix"

```

Dada una matriz `x` podemos darle vuelta (lo que se conoce como **matriz transpuesta**, donde filas pasan a ser columnas y viceversa) con la función `t()`.

```

x

##      edades      tlf      cp
## [1,]     14        NA 33007
## [2,]     24 683839390 28019
## [3,]     56 621539732 37005
## [4,]     31 618211286 18003
## [5,]     20        NA 33091
## [6,]     87 914727164 25073
## [7,]     73        NA 17140

t(x) # Matriz transpuesta

##      [,1]      [,2]      [,3]      [,4]      [,5]      [,6]      [,7]
## edades     14        24        56        31        20        87        73
## tlf        NA 683839390 621539732 618211286        NA 914727164        NA

```

```
## cp      33007      28019      37005      18003 33091      25073 17140
```

## 5.2 Apply vs bucles

Si has programado en algún otro lenguaje, estarás echando en falta elementos como un `if (blabla) {...} else {...}` (que los usaremos a veces) o bucles `for` y `while`.

**¿No existen los bucles en R?** Sí, sí existen. He aquí un ejemplo.

```
v <- rep(0, 20) # Vector de 20 ceros
for (i in 1:20) { # Bucle en base a un índice i que va de 1 a 20
  v[i] <- i^2 # En cada iteración guardamos el valor de i al cuadrado en el elemento i de v
}
v
## [1]  1  4  9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361
## [20] 400
```

La razón por la que no hemos usado aún **bucles**, e intentaremos evitarlos lo máximo posible, es porque son **muy inefficientes** en tiempo de ejecución. Vamos a poner un ejemplo. Supongamos que de la matriz `x` queremos calcular la suma de cada fila (es decir, 7 valores) o la suma de cada columna (3 valores). Con bucles sería así.

```

suma_por_filas <- rep(0, dim(x)[1]) # dim(x)[1] número de filas
suma_por_cols <- rep(0, dim(x)[2]) # dim(x)[2] número de columnas
for (i in 1:dim(x)[1]) { # Bucle recorriendo filas

  suma_por_filas[i] <- sum(x[i, ], na.rm = TRUE) # Suma de la fila i, eliminando NA

}

suma_por_filas

## [1] 33021 683867433 621576793 618229320 33111 914752324 17213

for (j in 1:dim(x)[2]) { # Bucle recorriendo filas

  suma_por_cols[j] <- sum(x[, j], na.rm = TRUE) # Suma de la columna j, eliminando NA

}

suma_por_cols

## [1] 305 2838317572 191338

```

En el código anterior puedes ver como para **acceder a la fila i-ésima de la matriz** se usa el operador `[i, ]` (dejando libre el índice de la columna), mientras que para **acceder a la columna j-ésima de la matriz** se usa el operador `[, j]`. Para **acceder al elemento (i, j)** se usa el operador `[i, j]`. También habrás observado que, si escribes los bucles en tu script, tienen una flecha a la izquierda para ser minimizados.

La pregunta es: **¿no hay una forma más eficiente de hacerlo?**

La respuesta: sí. La función `apply()` nos permite ejecutar una función por filas o por columnas.

```
# Suma por filas (MARGIN = 1) quitando NA
suma_por_filas <- apply(x, MARGIN = 1, FUN = "sum", na.rm = TRUE)
suma_por_filas

## [1] 33021 683867433 621576793 618229320 33111 914752324 17213

# Una función cualquiera por filas
operacion_por_filas <- apply(x, MARGIN = 1, FUN = function(x) { sum(sqrt(2) - 2) })
operacion_por_filas

## [1] -0.5857864 -0.5857864 -0.5857864 -0.5857864 -0.5857864 -
0.5857864 -0.5857864

# Suma por columnas (MARGIN = 2) quitando NA
suma_por_cols <- apply(x, MARGIN = 2, FUN = "sum", na.rm = TRUE)
suma_por_cols

##      edades       tlf        cp
## 305 2838317572 191338
```

Como puedes observar, necesitas tres argumentos y otros opcionales: la matriz, el índice por el que operar (`MARGIN = 1` por filas, `MARGIN = 2` por columnas) y la función a aplicar, amén de otros argumentos extras que pudiera necesitar la función.

Veamos qué es más eficiente con el ya conocido paquete `{microbenchmark}`.

```

microbenchmark::microbenchmark(for (i in 1:dim(x)[1]) {
  suma_por_filas[i] <- sum(x[i, ], na.rm = TRUE)}, apply(x, MARGIN = 1, FUN = "sum", na.rm = TRUE)

## Unit: microseconds
##                                         expr
## for (i in 1:dim(x)[1]) {    suma_por_filas[i] <- sum(x[i, ], na.rm = TRUE) }
##                               apply(x, MARGIN = 1, FUN = "sum", na.rm = TRUE)
##      min       lq     mean   median       uq      max neval cld
## 5265.692 7738.8675 10209.1556 9282.173 11805.926 25933.019 100   b
## 44.271 104.3165 139.5918 125.009 161.149 703.218 100   a

```

**¡El bucle nos tarda 50 veces más que el `apply`!** Di no a los bucles: casi siempre hay una forma mejor de hacerlo.

### 5.3 Tablas: data.frames

Además del nombre de las columnas que ha heredado la matriz `x` de la concatenación de las columnas que hemos realizado, podemos poner **nombre a los registros**, por ejemplo, el nombre de las personas a las que pertenece cada dato, definiendo una nueva variable con los nombres y concatenándola.

```

nombres <- c("Sonia", "Carla", "Pepito", "Carlos", "Lara", "Sandra", "Javi")
cbind(nombres, x)

##      nombres edades tlf          cp
## [1,] "Sonia"  "14"    NA        "33007"
## [2,] "Carla"  "24"  "683839390" "28019"

```

```
## [3,] "Pepito" "56"    "621539732" "37005"
## [4,] "Carlos"  "31"    "618211286" "18003"
## [5,] "Lara"    "20"    NA          "33091"
## [6,] "Sandra"  "87"    "914727164" "25073"
## [7,] "Javi"    "73"    NA          "17140"
```

### ¿Has visto lo que ha sucedido?

Como **una matriz SOLO puede tener un tipo de dato**, al añadir una variable de tipo textos, ha convertido los números también a texto poniéndole comillas: **hemos roto la integridad de nuestro dato**. Una forma de añadir **nombre a los registros, sin incluirlo como variable**, es usando la función `row.names()`.

```
row.names(x) <- c("Sonia", "Carla", "Pepito", "Carlos", "Lara", "Sandra", "Javi")
x
```

```
##      edades      tlf      cp
## Sonia     14      NA 33007
## Carla     24 683839390 28019
## Pepito    56 621539732 37005
## Carlos    31 618211286 18003
## Lara      20      NA 33091
## Sandra    87 914727164 25073
## Javi      73      NA 17140
```

*¿Qué sucede si realmente queremos añadir variables cuyos tipos sean distintos (¡ojo, pero con la misma longitud!)?*

Vamos a crear nuevas variables de texto `nombres` y `apellidos`, un valor lógico `casado` y una fecha `fecha_creacion` (fecha de entrada en el sistema) para cada persona.

```
# Nombres
nombres <- c("Sonia", "Carla", "Pepito", "Carlos", "Lara", "Sandra", "Javi")

# Apellidos
apellidos <- c("Pérez", "González", "Fernández", "Martínez", "Liébana", "García", "Ortiz")

# Estado civil (no lo sabemos de una persona)
casado <- c(TRUE, FALSE, FALSE, NA, TRUE, FALSE, FALSE)

# Fecha de creación (fecha en el que esa persona entra en el sistema)
# lo convertimos a tipo fecha
fecha_creacion <- as.Date(c("2021-03-04", "2020-10-12", "1990-04-05",
                           "2019-09-10", "2017-03-21", "2020-07-07",
                           "2000-01-28"))
```

Seguimos teniendo 7 registros, uno por persona pero ahora tenemos un popurrí de variables, de la **misma longitud pero de tipos distintos**:

- (`edades`, `tlf`, `cp`) son variables numéricas.
- (`nombres`, `apellidos`) son variables de texto.
- `casado` es una variable lógica.
- `fecha_creacion` de tipo fecha.

**¿Qué sucedería si yo intento mezclar todo en una matriz?**

```
# Juntamos todo en una matriz (juntamos por columnas)

x <- cbind(nombres, apellidos, edades, tlf, cp, casado, fecha_creacion)

##      nombres    apellidos   edades     tlf       cp     casado fecha_creacion
## [1,] "Sonia"    "Pérez"     "14"    NA      "33007"  "TRUE"  "18690"
## [2,] "Carla"    "González"  "24"    "683839390" "28019"  "FALSE"  "18547"
## [3,] "Pepito"   "Fernández" "56"    "621539732" "37005"  "FALSE"  "7399"
## [4,] "Carlos"   "Martínez"  "31"    "618211286" "18003"  NA      "18149"
## [5,] "Lara"     "Liébana"   "20"    NA      "33091"  "TRUE"  "17246"
## [6,] "Sandra"   "García"    "87"    "914727164" "25073"  "FALSE"  "18450"
## [7,] "Javi"     "Ortiz"     "73"    NA      "17140"  "FALSE"  "10984"
```

Efectivamente: como **en una matriz solo puede haber datos de un tipo**, los números los convierte a texto, las variables lógicas las convierte a texto ("TRUE" es un valor lógico, "TRUE" es un texto, como "Pepito", sin significado lógico - booleano - de verdadero/falso) y las fechas las ha convertido a texto (aunque las veas igual, ya no son de tipo de fecha, son texto y no podemos operar con ellas).

```
# Días entre la primera y el segundo elemento de fecha de creación

fecha_creacion[1] - fecha_creacion[2]
```

```
## Time difference of 143 days
```

```
# Días entre primera y segunda fecha de creación pero tomándolo de nuestra matriz (columna 7)

x[1, 7] - x[2, 7]
```

```
## Error in x[1, 7] - x[2, 7]: argumento no-numérico para operador binario
```

He aquí LA pregunta: **¿cómo juntar variables de distinto tipo, sin cambiar su naturaleza, como cuando juntamos datos en una tabla de excel?**

El formato de tabla de datos en R que vamos a empezar a usar se llama **data.frame**: una colección de variables de igual longitud pero cada una de un tipo distinto. Para crear un objeto de este tipo basta con usar la función **data.frame()**, pasándole como argumentos (separados por comas) las variables que queremos reunir, indicando en texto "..." el nombre de las columnas.

```
# Creamos nuestro primer data.frame

tabla <- data.frame("Nombre" = nombres, "Apellido" = apellidos,
                     "Edad" = edades, "Teléfono" = tlf,
                     "Código Postal" = cp, "Casado" = casado,
                     "Fecha_de_creación" = fecha_creacion)

tabla
```

	Nombre	Apellido	Edad	Teléfono	Código Postal	Casado	Fecha_de_creación
## 1	Sonia	Pérez	14	NA	33007	TRUE	2021-03-04
## 2	Carla	González	24	683839390	28019	FALSE	2020-10-12
## 3	Pepito	Fernández	56	621539732	37005	FALSE	1990-04-05
## 4	Carlos	Martínez	31	618211286	18003	NA	2019-09-10
## 5	Lara	Liébana	20	NA	33091	TRUE	2017-03-21
## 6	Sandra	García	87	914727164	25073	FALSE	2020-07-07
## 7	Javi	Ortiz	73	NA	17140	FALSE	2000-01-28

**¡TENEMOS NUESTRO PRIMER CONJUNTO DE DATOS!**

### 5.3.1 Data.frames: selección manual de columnas y filas

Si tenemos un `data.frame` ya creado y queremos **añadir una columna** es tan simple como usar la función `data.frame()` que ya hemos visto para concatenar la columna. Si queremos acceder a una columna, fila o elemento en concreto, **los data.frame tienen las mismas ventajas que una matriz**, así basta con usar los mismos operadores.

```
# Añadimos una nueva columna con nº de hermanos/as
hermanos <- c(0, 0, 1, 5, 2, 3, 0)
tabla <- data.frame(tabla, hermanos)
tabla

##  Nombre Apellido Edad Teléfono Código.Postal Casado Fecha_de_creación
## 1 Sonia Pérez 14 NA 33007 TRUE 2021-03-04
## 2 Carla González 24 683839390 28019 FALSE 2020-10-12
## 3 Pepito Fernández 56 621539732 37005 FALSE 1990-04-05
## 4 Carlos Martínez 31 618211286 18003 NA 2019-09-10
## 5 Lara Liébana 20 NA 33091 TRUE 2017-03-21
## 6 Sandra García 87 914727164 25073 FALSE 2020-07-07
## 7 Javi Ortiz 73 NA 17140 FALSE 2000-01-28
##   hermanos
## 1      0
## 2      0
## 3      1
```

```

## 4      5
## 5      2
## 6      3
## 7      0

# Accedemos a la tercera columna
tabla[, 3]

## [1] 14 24 56 31 20 87 73

# Accedemos a la quinta fila
tabla[5, ]

##  Nombre Apellido Edad Teléfono Código.Postal Casado Fecha_de_creación hermanos
## 5    Lara    Liébana    20        NA      33091    TRUE    2017-
03-21          2

# Accedemos a la tercera variable del quinto registro
tabla[5, 3]

## [1] 20

```

Un **data.frame** no solo tiene las ventajas de una matriz sino que también tiene las ventajas de una tabla de datos. Por ejemplo, podemos acceder a las variables por el índice de columna que ocupan pero también por su nombre, poniendo el nombre de la tabla, el símbolo \$ y con el tabulador nos aparecerá un menú de columnas a elegir.

### Volvamos a nuestro script.

Vamos a crear un script nuevo en la carpeta CODIGOS de nuestro proyecto que se

	Nombre	Apellido	Edad	Teléfono	Código.Postal	Casado	Fecha_de_creación	hermanos
1	Sonia	Pérez	14	NA	33007	TRUE	2021-03-04	0
2	Carl	Nombre	56	3990	28019	FALSE	2020-10-12	0
3	Pepito	Apellido	31	732	37005	FALSE	1990-04-05	1
4	Carlo	Edad	20	1286	18003	NA	2019-09-10	5
5	Lara	Teléfono	87	NA	33091	TRUE	2017-03-21	2
6	Sandra	Código.Postal	73	164	25073	FALSE	2020-07-07	3
7	Javi	Casado	73	NA	17140	FALSE	2000-01-28	0

Figure 5.1: Menú desplegable de variables (columnas) de un data.frame.

llame `primer_data_frame.R`. En él vamos a definir las variables que habíamos lanzado en consola, y vamos a construir el mismo `data.frame` llamado `tabla` que teníamos pero en nuestro script.

```

1 # Descripción: creación de nuestros primeros data.frame
2
3 # Variables
4 edades <- c(14, 24, 56, 31, 20, 87, 73) # vector numérico de longitud 7
5 tlf <- c(NA, 683839390, 621539732, 618211286, NA, 914727164, NA)
6 cp <- c(33007, 28019, 37005, 18003, 33091, 25073, 17140)
7 nombres <- c("Sonia", "Carla", "Pepito", "Carlos", "Lara", "Sandra", "Javi")
8 apellidos <- c("Pérez", "González", "Fernández", "Martínez", "Liébana", "García", "Ortiz")
9 casado <- c(TRUE, FALSE, FALSE, NA, TRUE, FALSE, FALSE)
10 fecha_creacion <- as.Date(c("2021-03-04", "2020-10-12", "1990-04-05", "2019-09-10",
11 "2017-03-21", "2020-07-07", "2000-01-28"))
12 hermanos <- c(0, 0, 1, 5, 2, 3, 0)
13
14 tabla <- data.frame(Nombre = nombres, Apellido = apellidos,
15 Edad = edades, Teléfono = tlf, Código.Postal = cp,
16 Casado = casado, Fecha_de_creación = fecha_creacion)
17
18 # Creamos el data.frame
19 tabla <- data.frame(Nombre = nombres, Apellido = apellidos,
20 Edad = edades, Teléfono = tlf, Código.Postal = cp,
21 Casado = casado, Fecha_de_creación = fecha_creacion)
22
23 # tabla <- data.frame(tabla, hermanos)
24
25 
```

Figure 5.2: Creando nuestro primer data.frame en el script.

```

# Descripción: creación de nuestros primeros data.frame

# Variables

edades <- c(14, 24, 56, 31, 20, 87, 73) # vector numérico de longitud 7

tlf <- c(NA, 683839390, 621539732, 618211286, NA, 914727164, NA)

cp <- c(33007, 28019, 37005, 18003, 33091, 25073, 17140)

nombres <- c("Sonia", "Carla", "Pepito", "Carlos", "Lara", "Sandra", "Javi")

apellidos <- c("Pérez", "González", "Fernández", "Martínez", "Liébana", "García", "Ortiz")

casado <- c(TRUE, FALSE, FALSE, NA, TRUE, FALSE, FALSE)

```

```

fecha_creacion <-
  as.Date(c("2021-03-04", "2020-10-12", "1990-04-05", "2019-09-10",
           "2017-03-21", "2020-07-07", "2000-01-28"))

hermanos <- c(0, 0, 1, 5, 2, 3, 0)

# Creamos el data.frame

tabla <- data.frame("Nombre" = nombres, "Apellido" = apellidos,
                     "Edad" = edades, "Teléfono" = tlf, "Código Postal" = cp,
                     "Casado" = casado, "Fecha_de_creación" = fecha_creacion)

tabla <- data.frame(tabla, hermanos)

tabla

##  Nombre Apellido Edad Teléfono Código.Postal Casado Fecha_de_creación
## 1 Sonia Pérez 14 NA 33007 TRUE 2021-03-04
## 2 Carla González 24 683839390 28019 FALSE 2020-10-12
## 3 Pepito Fernández 56 621539732 37005 FALSE 1990-04-05
## 4 Carlos Martínez 31 618211286 18003 NA 2019-09-10
## 5 Lara Liébana 20 NA 33091 TRUE 2017-03-21
## 6 Sandra García 87 914727164 25073 FALSE 2020-07-07
## 7 Javi Ortiz 73 NA 17140 FALSE 2000-01-28

##  hermanos
## 1 0
## 2 0
## 3 1
## 4 5

```

```

## 5      2
## 6      3
## 7      0

12 # Cargamos fichero variables.R
13 source("./CODIGOS/variables.R")
14
15 ######
16 # PRIMER USO DEL SCRIPT
17 # ■■■■■
30
31 ######
32 # PRIMEROS DATA.FRAME
33 ######
34 source("./CODIGOS/primer_data_frame.R")
35

```

Figure 5.3: Llamando a nuestro script desde nuestro código principal.

Además de dicho conjunto de datos, vamos a instalar (sino lo hemos hecho nunca en este ordenador) un paquete muy útil en R llamado `{datasets}`. Los paquetes que vayamos necesitando los instalaremos y llamaremos al inicio del código principal.

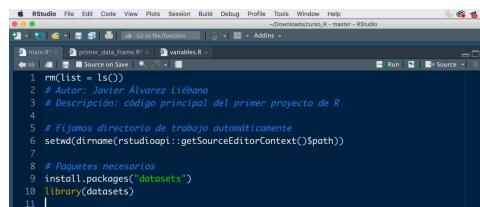


Figure 5.4: Instalamos y cargamos los paquetes necesarios al principio de nuestro main.R.

```

# Paquetes necesarios

# install.packages("datasets") # Descomentar si nunca se ha instalado

library(datasets)

```

Tras ello llamaremos a nuestro script `primer_data_frame.R` desde nuestro código principal `main.R` y guardaremos el script con el *source on save* activado para que se ejecute. Además de que ahora tenemos nuestro conjunto de datos `tabla` en nuestro panel de entorno, si escribimos `datasets::` y pulsamos tabulador, se nos

abre un desplegable con distintos conjuntos de datos para ser usados: el paquete `datasets` nos proporciona `data.frames` de prueba para que podamos usarlos en nuestros códigos según vamos aprendiendo.

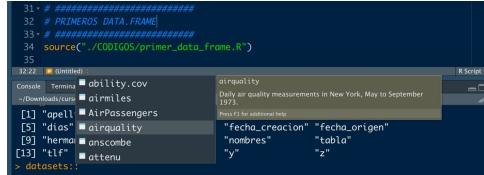


Figure 5.5: Menú desplegable con los data.frame de prueba en datasets

Una de las **ventajas de los data.frame** es que podemos visualizarlos como si fuera una tabla de Excel dentro de nuestro R con la función `View()`. Vamos a visualizar no solo el conjunto de datos `tabla` sino también el conjunto `iris` del paquete `datasets`: **los data.frame nos permiten trabajar con ellos como bases de datos o como matrices, con las ventajas de ambos.**

```
View(iris)
View(tabla)
```

The screenshot shows an RStudio interface. The code editor has two lines of code: `View(iris)` and `View(tabla)`. Below the code editor, the R console shows the output of these commands. The 'iris' dataset is displayed as a table with columns: Nombre, Apellido, Edad, Telefono, Código.Postal, Casado, Fecha\_de\_creación, and hermanos. The 'tabla' dataset is also displayed as a table with the same columns. At the bottom of the console, the following command history is visible:

```

Showing 1 to 7 of 7 entries, 8 total columns
[9] "hermanos"      "hoy"           "nombres"       "tabla"
[13] "tfl"           "x"              "y"              "z"
> View(iris)
> View(tabla)
>

```

Figure 5.6: Menú desplegable con los data.frame de prueba en datasets

En el caso de los `data.frame` tenemos además a nuestro disposición una **heramienta muy potente: la función `subset()`** Dicha función nos va a permitir seleccionar filas y columnas automáticamente, tomando de entrada los siguientes argumentos

- `x`: una tabla de entrada, un `data.frame` de entrada.
- `subset`: la condición lógica que queramos usar para seleccionar registros (filas).
- `select`: un vector que contenga el nombre de las columnas que queremos seleccionar (a lo mejor solo queremos filtrar por filas pero quizás también por columnas).

Por ejemplo, vamos a seleccionar solo los nombres y apellidos de aquellas personas mayores de edad de nuestro conjunto de datos `tabla`, y del conjunto `iris` vamos a extraer todos los registros en los que el largo del sépalo es mayor que 7.1, seleccionando solo las columnas de longitud de sépalo y la especie de la planta.

```
subset(tabla, subset = Edad > 18, select = c("Nombre", "Apellido"))
```

```
##    Nombre Apellido
## 2  Carla González
## 3 Pepito Fernández
## 4 Carlos Martínez
## 5  Lara Liébana
## 6 Sandra García
## 7    Javi Ortiz
```

```
subset(iris, subset = Sepal.Length > 7.1, select = c("Sepal.Length", "Species"))

##      Sepal.Length   Species
## 106          7.6 virginica
## 108          7.3 virginica
## 110          7.2 virginica
## 118          7.7 virginica
## 119          7.7 virginica
## 123          7.7 virginica
## 126          7.2 virginica
## 130          7.2 virginica
## 131          7.4 virginica
## 132          7.9 virginica
## 136          7.7 virginica
```

## 5.4 Consejos y tips

Acceso a librería

A veces puede que no queramos cargar todo un paquete sino solo una función del mismo, para lo que es suficiente `nombre_paquete::nombre_funcion`.

Nombre de variables

La función `names()` no solo sirve para consultar los nombres de las variables de un `data.frame` sino también para cambiarlos a nuestro gusto.

```

# Consultamos nombres

names(tabla)

## [1] "Nombre"          "Apellido"        "Edad"
## [4] "Teléfono"        "Código.Postal"   "Casado"
## [7] "Fecha_de_creación" "hermanos"

# Cambiamos nombres

names(tabla) <- c("nombre_persona", "apellido_persona", "edad", "tlf",
                  "cp", "casado", "f_creacion")

tabla

## nombre_persona apellido_persona edad      tlf      cp casado f_creacion NA
## 1      Sonia       Pérez     14      NA 33007  TRUE 2021-03-04  0
## 2      Carla       González    24 683839390 28019 FALSE 2020-
## 10-12  0
## 3      Pepito      Fernández    56 621539732 37005 FALSE 1990-
## 04-05  1
## 4      Carlos      Martínez    31 618211286 18003      NA 2019-09-10  5
## 5      Lara        Liébana    20      NA 33091  TRUE 2017-03-21  2
## 6      Sandra      García     87 914727164 25073 FALSE 2020-
## 07-07  3
## 7      Javi        Ortiz     73      NA 17140 FALSE 2000-01-28  0

```

Paquete {tibble}

En dicho paquete tienes más **funciones para una gestión más ágil, eficiente y coherente de los `data.frame`**. Ver <https://tibble.tidyverse.org/>.

## 5.5 □ Ejercicios

Ejercicio 1: define una matriz de ceros de 3 filas y 7 columnas. Tras hacerlo calcula su transpuesta y obtén sus dimensiones

- Solución:

```
# Matriz

matriz <- matrix(0, nrow = 3, ncol = 7)

# Transpuesta

t(matriz)

##      [,1] [,2] [,3]
## [1,]     0     0     0
## [2,]     0     0     0
## [3,]     0     0     0
## [4,]     0     0     0
## [5,]     0     0     0
## [6,]     0     0     0
## [7,]     0     0     0

# Dimensiones transpuesta

dim(t(matriz))
```

```
## [1] 7 3
```

```
ncol(t(matriz))
```

```
## [1] 3
```

```
nrow(t(matriz))
```

```
## [1] 7
```

Ejercicio 2: calcula la suma de cada fila de la matriz `matriz <- matrix(1:12, nrow = 4)` usando un bucle. Haz lo mismo evitando usar bucles.

- Solución:

```
# Matriz

matriz <- matrix(1:12, nrow = 4)

# Con bucle (recorremos sus filas)
suma <- rep(0, nrow(matriz)) # Definimos un vector de 0's con tantos elementos como filas

for (i in 1:nrow(matriz)) {

    suma[i] <- sum(matriz[i, ]) # Sumamos la fila i
}

suma

## [1] 15 18 21 24
```

```
# Sin bucle (MARGIN = 1 ya que es una operación por filas)

suma <- apply(matriz, MARGIN = 1, FUN = "sum")

suma

## [1] 15 18 21 24
```

Ejercicio 3: del conjunto `iris` del paquete `datasets` obtén el nombre de las variables, y selecciona aquellas filas cuya variable `Petal.Width` sea distinta de 0.2, y quédate solo con las variables `Sepal.Length`, `Sepal.Width` y `Species`. Calcula el número de filas borradas. Tras hacer todo ello, traduce a castellano el nombre de las columnas del `data.frame` filtrado.

- Solución:

```
# Nombres de variables

names(iris)

## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"

# Filtramos filas

filtro_fila <- subset(iris, subset = Petal.Width != 0.2)

# Filtramos columnas

filtro_col <- subset(filtro_fila, select = c("Sepal.Length", "Sepal.Width", "Species"))

# Todo de una vez

filtro <- subset(iris, subset = Petal.Width != 0.2, select = c("Sepal.Length", "Sepal.Width", "Spe
```

```
filtro
```

	##	Sepal.Length	Sepal.Width	Species
	## 6	5.4	3.9	setosa
	## 7	4.6	3.4	setosa
	## 10	4.9	3.1	setosa
	## 13	4.8	3.0	setosa
	## 14	4.3	3.0	setosa
	## 16	5.7	4.4	setosa
	## 17	5.4	3.9	setosa
	## 18	5.1	3.5	setosa
	## 19	5.7	3.8	setosa
	## 20	5.1	3.8	setosa
	## 22	5.1	3.7	setosa
	## 24	5.1	3.3	setosa
	## 27	5.0	3.4	setosa
	## 32	5.4	3.4	setosa
	## 33	5.2	4.1	setosa
	## 38	4.9	3.6	setosa
	## 41	5.0	3.5	setosa
	## 42	4.5	2.3	setosa
	## 44	5.0	3.5	setosa
	## 45	5.1	3.8	setosa
	## 46	4.8	3.0	setosa
	## 51	7.0	3.2	versicolor

```
## 52      6.4      3.2 versicolor
## 53      6.9      3.1 versicolor
## 54      5.5      2.3 versicolor
## 55      6.5      2.8 versicolor
## 56      5.7      2.8 versicolor
## 57      6.3      3.3 versicolor
## 58      4.9      2.4 versicolor
## 59      6.6      2.9 versicolor
## 60      5.2      2.7 versicolor
## 61      5.0      2.0 versicolor
## 62      5.9      3.0 versicolor
## 63      6.0      2.2 versicolor
## 64      6.1      2.9 versicolor
## 65      5.6      2.9 versicolor
## 66      6.7      3.1 versicolor
## 67      5.6      3.0 versicolor
## 68      5.8      2.7 versicolor
## 69      6.2      2.2 versicolor
## 70      5.6      2.5 versicolor
## 71      5.9      3.2 versicolor
## 72      6.1      2.8 versicolor
## 73      6.3      2.5 versicolor
## 74      6.1      2.8 versicolor
## 75      6.4      2.9 versicolor
## 76      6.6      3.0 versicolor
```

## 77	6.8	2.8 versicolor
## 78	6.7	3.0 versicolor
## 79	6.0	2.9 versicolor
## 80	5.7	2.6 versicolor
## 81	5.5	2.4 versicolor
## 82	5.5	2.4 versicolor
## 83	5.8	2.7 versicolor
## 84	6.0	2.7 versicolor
## 85	5.4	3.0 versicolor
## 86	6.0	3.4 versicolor
## 87	6.7	3.1 versicolor
## 88	6.3	2.3 versicolor
## 89	5.6	3.0 versicolor
## 90	5.5	2.5 versicolor
## 91	5.5	2.6 versicolor
## 92	6.1	3.0 versicolor
## 93	5.8	2.6 versicolor
## 94	5.0	2.3 versicolor
## 95	5.6	2.7 versicolor
## 96	5.7	3.0 versicolor
## 97	5.7	2.9 versicolor
## 98	6.2	2.9 versicolor
## 99	5.1	2.5 versicolor
## 100	5.7	2.8 versicolor
## 101	6.3	3.3 virginica

```
## 102      5.8      2.7  virginica
## 103      7.1      3.0  virginica
## 104      6.3      2.9  virginica
## 105      6.5      3.0  virginica
## 106      7.6      3.0  virginica
## 107      4.9      2.5  virginica
## 108      7.3      2.9  virginica
## 109      6.7      2.5  virginica
## 110      7.2      3.6  virginica
## 111      6.5      3.2  virginica
## 112      6.4      2.7  virginica
## 113      6.8      3.0  virginica
## 114      5.7      2.5  virginica
## 115      5.8      2.8  virginica
## 116      6.4      3.2  virginica
## 117      6.5      3.0  virginica
## 118      7.7      3.8  virginica
## 119      7.7      2.6  virginica
## 120      6.0      2.2  virginica
## 121      6.9      3.2  virginica
## 122      5.6      2.8  virginica
## 123      7.7      2.8  virginica
## 124      6.3      2.7  virginica
## 125      6.7      3.3  virginica
## 126      7.2      3.2  virginica
```

## 127	6.2	2.8	virginica
## 128	6.1	3.0	virginica
## 129	6.4	2.8	virginica
## 130	7.2	3.0	virginica
## 131	7.4	2.8	virginica
## 132	7.9	3.8	virginica
## 133	6.4	2.8	virginica
## 134	6.3	2.8	virginica
## 135	6.1	2.6	virginica
## 136	7.7	3.0	virginica
## 137	6.3	3.4	virginica
## 138	6.4	3.1	virginica
## 139	6.0	3.0	virginica
## 140	6.9	3.1	virginica
## 141	6.7	3.1	virginica
## 142	6.9	3.1	virginica
## 143	5.8	2.7	virginica
## 144	6.8	3.2	virginica
## 145	6.7	3.3	virginica
## 146	6.7	3.0	virginica
## 147	6.3	2.5	virginica
## 148	6.5	3.0	virginica
## 149	6.2	3.4	virginica
## 150	5.9	3.0	virginica

```
# Filas borradas  
nrow(iris) - nrow(filtro)  
  
## [1] 29  
  
# Cambiamos nombres a castellano del conjunto filtrado  
names(filtro) <- c("longitud_sepalo", "anchura_sepalo", "especies")
```



# Chapter 6

## Importando/exportando

Hemos aprendido a **crear nuestros propios datos** pero la mayoría de veces los cargaremos de distintos archivos, fuentes, etc. Vamos a ver las **4 formas más comunes de importar (cargar) datos**

### 6.1 Importación de datos

Las **4 formas más comunes de importar (cargar) datos** son:

- desde un archivo propio de R (extensión .RData).
- desde un archivo separado por comas (un archivo .csv).
- desde un excel (archivo .xlsx).
- desde un enlace de internet.

### 6.1.1 Archivo .RData

La forma más sencilla de guardar datos y variables en R, y que además ocupa menos espacio en nuestro disco duro, es guardarlos en archivos propios que tiene R como son los archivos con extensiones `.rda` y `.RData`.

Es **recomendable** tener los datos en la misma carpeta del proyecto pero una carpeta separada, ya que podemos tener muchos archivos y así no mezclamos dichos ficheros con los códigos que escribimos. En la carpeta `DATOS` del proyecto tenemos 4 archivos `.RData`: `coches.RData`, `panel_vacunas_ccaa.RData`, `panel_vacunas_fecha.RData` y `panel_variables.RData`. **¿Cómo cargar archivos .RData?**

Muy sencillo: como son ficheros nativos de R, basta con usar la función de carga `load()`, y dentro la ruta de los archivos.

```
# Al fijar directorio de trabajo, no necesitamos toda la ruta, solo "./" y la ruta dentro de la carpeta DATOS
load("./DATOS/coches.RData")
load("./DATOS/panel_vacunas_ccaa.RData")
load("./DATOS/panel_vacunas_fecha.RData")
load("./DATOS/panel_variables.RData")
```

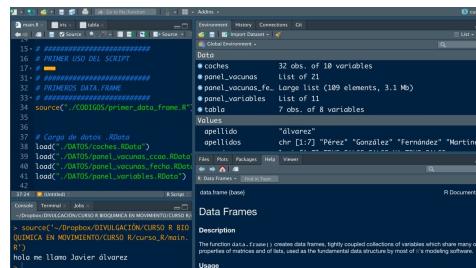


Figure 6.1: Importación de ficheros de extensión `.RData`.

Como ves en la imagen 6.1, en el panel de entorno de la parte superior derecha ahora tendremos 4 data.frames que antes no teníamos. Un función muy útil es `head()`, con argumento el nombre de un data.frame, que nos permite visualizar las primeras columnas.

```
# Ver las primeras filas de la tabla que guarda distintos modelos de coche y características
head(coches)

##                                     consumo cilindrada      peso    potencia     tiempo
## Mazda RX4           11.20069   2621.936 1188.411 111.52570 40.91298
## Mazda RX4 Wag       11.20069   2621.936 1304.077 111.52570 42.30491
## Datsun 710          10.31643   1769.807 1052.333  94.28991 46.25702
## Hornet 4 Drive      10.99134   4227.872 1458.298 111.52570 48.32006
## Hornet Sportabout   12.57832   5899.356 1560.356 177.42725 42.30491
## Valiant              12.99528   3687.097 1569.428 106.45635 50.25883
##                                     motor transmision ncyl ncarb ngear
## Mazda RX4            cilindros en V      Manual     6      4      4
## Mazda RX4 Wag        cilindros en V      Manual     6      4      4
## Datsun 710           cilindros en serie  Manual     4      1      4
## Hornet 4 Drive       cilindros en serie Automática  6      1      3
## Hornet Sportabout    cilindros en V      Automática  8      2      3
## Valiant               cilindros en serie Automática  6      1      3
```

### 6.1.2 Archivo .csv

Otra opción de importación habitual son los **archivos .csv (comma separated values)**: son archivos separados por comas (u otro carácter como puntos, puntos y co-

mas, o tabuladores). En apariencia cuando los abrimos en el ordenador son como un Excel (ya que los abre el Excel), pero ocupan mucho menos que un Excel y su **lectura es universal** (independiente de tener instalado o no el Excel) ya que son archivos de texto **sin formato**.

Para leer un archivo .csv basta con usar la función `read.csv()`, y la mayoría de las veces basta indicarle la ruta del archivo para su lectura (argumento `file`).

```
vacunas_esp <- read.csv(file = "./DATOS/datos_ES.csv")
```

```
names(vacunas_esp) # Todas las variables del data.frame
```

```
## [1] "fechas"  
## [2] "ISO"  
## [3] "poblacion"  
## [4] "porc_pobl_total"  
## [5] "poblacion_mayor_16a"  
## [6] "porc_pobl_total_mayor_16a"  
## [7] "dosis_entrega_pfizer"  
## [8] "dosis_entrega_astra"  
## [9] "dosis_entrega_moderna"  
## [10] "dosis_entrega_janssen"  
## [11] "dosis_entrega"  
## [12] "dosis_entrega_100hab"  
## [13] "porc_entregadas_sobre_total"  
## [14] "dosis_diarias_entrega_pfizer"  
## [15] "dosis_diarias_entrega_astra"
```

```
## [16] "dosis_diarias_entrega_moderna"
## [17] "dosis_diarias_entrega"
## [18] "dosis_7D_entrega_pfizer"
## [19] "dosis_7D_entrega_astra"
## [20] "dosis_7D_entrega_moderna"
## [21] "dosis_7D_entrega"
## [22] "dosis_7D_entrega_100hab"
## [23] "dosis_admin"
## [24] "dosis_primera"
## [25] "dosis_pauta_completa"
## [26] "dosis_admin_100hab"
## [27] "porc_admin_sobre_ccaa"
## [28] "porc_admin_vs_total"
## [29] "dosis_diarias_admin"
## [30] "dosis_diarias_admin_100hab"
## [31] "crec_diario_dosis_admin"
## [32] "dosis_diarias_primera"
## [33] "dosis_diarias_segunda"
## [34] "dosis_7D_admin"
## [35] "dosis_7D_admin_100hab"
## [36] "crec_7D_dosis_admin"
## [37] "porc_admin_vs_total_7D"
## [38] "personas_vacunadas"
## [39] "personas_pauta_completa"
## [40] "personas_1dosis"
```

```
## [41] "porc_personas_vacunadas"
## [42] "porc_personas_pauta_completa"
## [43] "porc_personas_vacunadas_16a"
## [44] "porc_personas_pauta_completa_16a"
## [45] "personas_vacunadas_diarias"
## [46] "personas_pauta_completa_diarias"
## [47] "porc_personas_vacunadas_diarias"
## [48] "porc_personas_pauta_completa_diarias"
## [49] "personas_vacunadas_7D"
## [50] "personas_pauta_completa_7D"
## [51] "porc_personas_vacunadas_7D"
## [52] "porc_personas_pauta_completa_7D"
## [53] "porc_personas_vacunadas_16a_7D"
## [54] "porc_personas_pauta_completa_16a_7D"
## [55] "crec_diario_personas_vacunadas"
## [56] "crec_7D_personas_vacunadas"
## [57] "crec_diario_personas_pauta_completa"
## [58] "crec_7D_personas_pauta_completa"
## [59] "desv_porc_admin_vs_total"
## [60] "desv_dosis_entrega"
## [61] "desv_porc_personas_vacunadas"
## [62] "desv_porc_personas_pauta_completa"
## [63] "fecha_30vacunados ritmo7D"
## [64] "fecha_50vacunados ritmo7D"
## [65] "fecha_70vacunados ritmo7D"
```

```

## [66] "fecha_30inmunizados Ritmo7D"
## [67] "fecha_50inmunizados Ritmo7D"
## [68] "fecha_70inmunizados Ritmo7D"
## [69] "fecha_30inmunizados_16a Ritmo7D"
## [70] "fecha_50inmunizados_16a Ritmo7D"
## [71] "fecha_70inmunizados_16a Ritmo7D"

vacunas_esp[1:5, 1:7] # Primeras filas y columnas

##      fechas ISO poblacion porc_pobl_total poblacion_mayor_16a
## 1 2021-01-05 ES 47450795          100        40129822
## 2 2021-01-06 ES 47450795          100        40129822
## 3 2021-01-07 ES 47450795          100        40129822
## 4 2021-01-08 ES 47450795          100        40129822
## 5 2021-01-09 ES 47450795          100        40129822

##      porc_pobl_total_mayor_16a dosis_entrega_pfizer
## 1                      100          743925
## 2                      100          743925
## 3                      100          743925
## 4                      100          743925
## 5                      100          743925

```

Podemos personalizar la lectura en función del archivo, dando valores a los argumentos `sep` (para indicar el carácter que se está usando para separar columnas, en caso de no ser `,` por defecto), `dec` (el carácter que estamos usando para marcar decimales, por defecto es `.`) o `header` (por defecto en `TRUE`, lo que le indicamos que el nombre de las columnas está en la primera fila). Al tener nuestro archivo

preparado para que sirva con los parámetros por defecto no debemos añadirlo, pero podemos hacer la prueba para ver que la lectura es la misma.

```
vacunas_esp <- read.csv(file = "./DATOS/datos_ES.csv", sep = ",", dec = ".", header = TRUE)

vacunas_esp[1:5, 1:7] # Primeras filas y columnas
```

	## fechas	ISO	poblacion	porc_pobl_total	poblacion_mayor_16a	
## 1	2021-01-05	ES	47450795	100	40129822	
## 2	2021-01-06	ES	47450795	100	40129822	
## 3	2021-01-07	ES	47450795	100	40129822	
## 4	2021-01-08	ES	47450795	100	40129822	
## 5	2021-01-09	ES	47450795	100	40129822	
						## porc_pobl_total_mayor_16a dosis_entrega_pfizer
## 1				100	743925	
## 2				100	743925	
## 3				100	743925	
## 4				100	743925	
## 5				100	743925	

### 6.1.3 Archivo .xlsx

Muchas veces no tendremos un .csv (por desgracia) y nos tocará leer desde un excel. Para ello deberemos instalar (la primera vez) y cargar el paquete {readxl} que nos permitirá usar funciones para cargar archivos .xls (la función `read_xls()`) y archivos .xlsx (la función `read_xlsx()`). Además del argumento `path` con la ruta del archivo, podemos en el argumento `sheet` indicarle la hoja de Excel a leer (en caso de tener varias).

```
install.packages("readxl")
library(readxl)
boston <- read_xlsx(path = "./DATOS/Boston.xlsx")

head(boston)

## # A tibble: 6 × 14
##      crim    zn indus chas   nox    rm   age   dis   rad tax ptratio black
##      <dbl> <dbl>
## 1 0.00632    18  2.31    0 0.538  6.58  65.2  4.09    1  296  15.3  397.
## 2 0.0273     0  7.07    0 0.469  6.42  78.9  4.97    2  242  17.8  397.
## 3 0.0273     0  7.07    0 0.469  7.18  61.1  4.97    2  242  17.8  393.
## 4 0.0324     0  2.18    0 0.458  7.00  45.8  6.06    3  222  18.7  395.
## 5 0.0690     0  2.18    0 0.458  7.15  54.2  6.06    3  222  18.7  397.
## 6 0.0298     0  2.18    0 0.458  6.43  58.7  6.06    3  222  18.7  394.
## # ... with 2 more variables: lstat <dbl>, medv <dbl>
```

#### 6.1.4 Desde web

Por último, muchas veces querremos cargar archivos colgados en la web que, aunque al descargarlos son .csv o .xlsx, son archivos dinámicos que sabemos que van a ir cambiando, como por ejemplo los datos de casos covid, hospitalizados, ingresos UCI y fallecidos, de la página del ISCIII <https://cnecovid.isciii.es/covid19/#documentaci%C3%B3n-y-datos>.

Esos archivos cambian cada día, por lo que para visualizarlos, analizarlos o guardarlos cada día, tendríamos que, cada día, entrar de forma manual a la

The screenshot shows a section titled "Documentación y datos" under the heading "COVID-19 Distribución geográfica Evolución pandemia". It lists several CSV files available for download:

- casos\_tecnica\_ccaa.csv: Número de casos por técnica diagnóstica y CCAA (de residencia)
- casos\_tecnica\_ccaa\_res.csv: Número de casos por técnica diagnóstica y provincia (de residencia)
- casos\_diag\_ccaa.csv: Número de casos por fecha de diagnóstico (fecha\_diag), con la información detallada en el documento metadata\_diag\_ccaa\_deci\_prov\_edad\_sexo.pdf. Por favor, consulte este documento antes de la consulta de los datos del CSV.
- casos\_diag\_ccaa\_res.csv: Número de casos por fecha de diagnóstico (fecha\_diag) y CCAA (de residencia)
- casos\_hosp\_uci\_res.csv: Número de hospitalizaciones, número de ingresos en UCI y número de defunciones por sexo, edad y provincia de residencia.

**Nota informativa en relación a la notificación de la información de vigilancia a la RENAVE**

Los datos publicados en el Panel COVID-19 proceden de la declaración individualizada de casos COVID-19 a la Red Nacional de Vigilancia Epidemiológica (RENAVE) a través de la aplicación informática SIVEs. La COVID-19 es una enfermedad de declaración obligatoria y, como tal, la responsabilidad de la notificación de acuerdo a los criterios establecidos en cada momento corresponde a los facultativos y servicios de Salud Pública que realizan la notificación. Los datos que reflejan esta notificación se publican en el Panel COVID-19.

En SIVEs se contabiliza **toda** los casos notificados, siguiendo la estrategia de vigilancia vigente en cada momento (Estrategia de detección precoz, vigilancia y control de COVID-19 disponible en <https://www.sive.es/gestion/covid-19/estrategia-de-deteccion-precoz-vigilancia-y-control-de-covid-19.pdf>).

La información recogida por la vigilancia epidemiológica es distinta a cualquier información que se obtiene con fines estadísticos. Se recoge y usa, en tanto red, para la toma de decisiones en el ámbito de la salud pública. Por este motivo puede ser incompleta, contener errores y sufrir retrasos en distinta medida. En definitiva, se precisa de un tiempo para su depuración y consolidación. Para elaborar los informes publicados por el ISCIII se realizan procesos de depuración, imputación y relación con otras bases de datos secundarias para paliar los defectos de la notificación. Los análisis de una base entrada un día pueden variar de los extractos al día siguiente, ya que se han actualizado los datos.

Es importante resaltar que todos los resultados son provisionales y deben interpretarse con precaución porque se ofrece la información disponible en el momento de la extracción de datos. En los casos en los que no se haya notificado la provincia de residencia, ésta aparecerá en blanco. Es por tanto posible que no coincida la provincia de los casos notificados por provincias con el dato de la Comunidad Autónoma al que pertenezcan.

El número de enero de 2020 es dinámicamente debido a que es la cifra más reciente de datos de vigilancia en la plataforma SIVEs. Se revisan y actualizan conforme se confirmen por las CCAA.

Todo ello debe tenerse en cuenta a la hora de interpretar los datos globales que se ofrecen en este Panel COVID-19.

Figure 6.2: Archivos de la pandemia en el ISCIII.

página y bajarnos el archivo. O no...

R nos permite leer archivos subidos en una web, dándole a la función de lectura el enlace del archivo en lugar de la ruta local de nuestro ordenador (para averiguar el enlace, basta con clickar botón derecho en la web y seleccionar «copiar dirección de enlace»)

```
datos_ISCIII <- read.csv(file = "https://cnecovid.isciii.es/covid19/resources/casos_hosp.csv")

head(datos_ISCIII)

##  provincia_iso sexo grupo_edad      fecha num_casos num_hosp num_uci num_def

## 1             A   H 0-9 2020-01-01       0       0       0       0
## 2             A   H 10-19 2020-01-01      0       0       0       0
## 3             A   H 20-29 2020-01-01      0       0       0       0
## 4             A   H 30-39 2020-01-01      0       0       0       0
## 5             A   H 40-49 2020-01-01      0       0       0       0
## 6             A   H 50-59 2020-01-01      0       0       0       0
```

Mientras el enlace web no cambie, cada vez que ejecutemos esa orden en nuestro

código tendremos en `datos_ISCIII` el último archivo actualizado que haya, sea el que sea, sin tener que descargarlo de forma manual, ¡y sin necesidad de guardarlo en nuestro local, solo en la memoria virtual de nuestra sesión de R!

## 6.2 Exportación de datos

Aunque se puede exportar en cualquier formato que puedas importar, vamos a ver las **dos formas más útiles y eficientes de exportar datos en R**:

- fichero `.RData`.
- fichero `.csv` (obviaremos la exportación a Excel porque un `.csv` ya es posible abrirlo con dicho engendro del demonio).

### 6.2.1 Guardar en `.RData`

La exportación en fichero `.RData` es la opción **más recomendable si tú o tu equipo solo trabajáis con R**, es la opción nativa de fichero, para que su importación sea tan sencilla como una función `load()`. Para exportar en `R.Data` basta con uses la función `save()`, indícadole lo que quieras guardar y la ruta donde quieras guardarlo.

Es **importante** entender que la principal ventaja de exportar un fichero `.RData` es que no se está portando una tabla, o un fichero tabulado con un formato de filas y columnas: estás exportando **cualquier cosa**, cualquier variable de R, con la naturaleza de esa variable intacta, sin necesidad de pasarlo otro formato.

```
# Exportamos en .RData la variable nombres  
save(nombres, file = "./EXPORTAR/nombres.RData")
```

Para tenerlo organizado, la orden anterior está hecha habiendo creado en nuestra

carpeta del proyecto una carpeta EXPORTAR para guardar lo que vayamos exportando. Ese fichero solo podrá ser abierto por R, pero cuando lo cargemos, tendremos la variable nombres tal cual la hemos guardado.

### 6.2.2 Guardar en .csv

No siempre trabajamos en R y a veces necesitamos una exportación de un `data.frame` o una tabla que podamos abrir en nuestra ordenador, ya sea para explicársela a alguien o para enviársela a otra persona. Para ello exportaremos en `.csv`, un fichero sin formato, y que es capaz de ser abierto por todo tipo de hojas de cálculo: basta que usemos la función `write.csv()`.

```
# Exportamos en .csv el data.frame tabla
write.csv(tabla, file = "./EXPORTAR/tabla.csv")
```

Podemos consultar con `? write.csv` las distintas opciones de exportaciones (por ejemplo, con `row.names` podemos indicarle si queremos nombres de filas o no, y con `col.names` si queremos exportar la cabecera con el nombre de las columnas).

## 6.3 Consejos y tips

Paquetes `{readr}` y `{tidyverse}`

En dichos paquetes tienes más funciones para una fácil **exportación y tabulación de distintos tipos de datos**, sea el formato que sea. Ver <https://tidyverse.org/> y <https://readr.tidyverse.org/>



Figure 6.3: Paquete readr.



Figure 6.4: Paquete tidyverse.

### Paquete {rvest}

En dicho paquete tienes más **funciones para una lectura directamente de una página web (no desde un documento, desde la propia página web, como si estuvieras navegando en ella)**. Ver <https://github.com/tidyverse/rvest>.

### Secciones en el código

Los comentarios no solo sirven para documentar el código sino que además pueden servirnos para **construir secciones de código**. Prueba a escribir un comentario con varias #####: ¿ves la flecha que te aparece en la parte izquierda? Sirve para minimizar o maximizar trozos de código, de forma que tu código aún más limpio.

### Líneas de código en los errores

Dado que los errores del código nos vendrán referenciados en la consola por el

```

1+ # #####
2# PRIMER USO DEL SCRIPT
3# #####
4
5# Sumamos 3 a cada elemento de x
6z <- x + 3
7
8# Imprimimos por pantalla la frase unida, y a dicha frase
9# le pegamos nuestro apellido
10cat(paste(paste(y, collapse = " "), apellido))
11
12# Dias que han pasado desde el inicio de año
13dias <- hoy - fecha_origen

```

Figure 6.5: Secciones en el código.

número de línea donde fueron detectados, puede sernos muy útil mostrar dichos números en la barra lateral izquierda, yendo a `Tools << Global Options << Code << Display << Show line numbers`

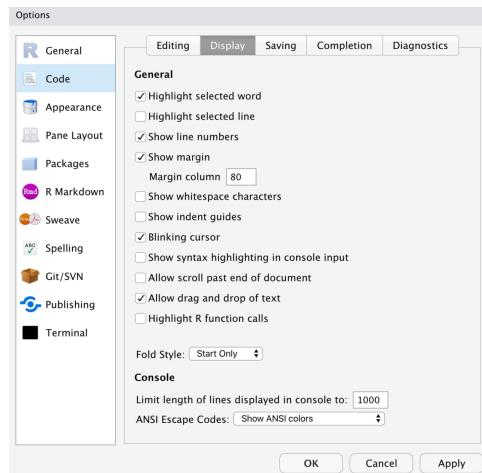


Figure 6.6: Líneas de código.

### Margen derecho en la ventana de scripts

Aunque no afecte a nuestro código escribir todo en una línea sin saltos de línea, no somos bárbaros/as. ¿Por qué cuadno escribes en un Word lo haces en formato vertical pero cuando programas pones todas las órdenes seguidas? Recuerda que

la legibilidad de tu código no solo te ahorrará tiempo sino que te hará programar mejor. ¿Cómo podemos fijar un margen imaginario para nosotros ser quienes debemos al *ENTER*? Yendo a `Tools << Global Options << Code << Display << Show margin` (es un margen imaginario para ser nosotros quienes lo hagamos efectivo, a R le da igual)

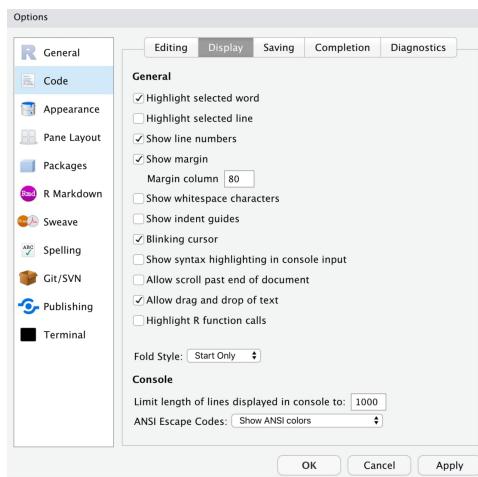


Figure 6.7: Margen derecho.



## Chapter 7

# Estructuras de control

Aunque la mayoría de veces son sustituibles por otras expresiones más legibles y eficientes, es importante que conozcamos como se usan las **expresiones de control más típicas**.

### 7.1 if...else

Como en cualquier lenguaje de programación, las **estructuras if...else...** nos permiten ejecutar partes de nuestro código solo cuando se cumple la condición o condiciones lógicas que queremos. Cuando ejecutamos un `if` estamos diciendo

SI las condiciones impuestas se cumplen (TRUE), ejecuta las ordenes que tengamos dentro de las llaves. En caso contrario, no sucede nada SALVO que tengamos además un `else` anidado.

Si el `if` devuelve FALSE, no sucederá nada **SALVO** que tengamos además un `else`:

**lo que sucede cuando no se cumple**, como en el ejemplo que tenemos debajo.

```
edades <- c(14, 24, 56, 31, 20, 87, 73)
mayores_de_edad <- FALSE

if (all(edades >= 18)) { # Si todas las personas son mayores de edad

  mayores_de_edad <- TRUE
  print("todos mayores de edad")

} else { # si la condición no se cumple: filtramos solo los mayores

  edades <- edades[edades >= 18]
  print("algún menor de edad se ha eliminado")

}

## [1] "algún menor de edad se ha eliminado"
```

Las órdenes dentro del primer `if` se ejecutarán si todas las personas son mayores de edad. En caso de no cumplirse (como es el caso), se ejecutarán lo que hay entre llaves tras el `else` (filtra solo los mayores de edad e imprime un mensaje de control).

Dicha estructura puede **anidarse**, de forma que vayamos concatenando estructuras `if else`, como en el ejemplo que tenemos debajo.

```
edades <- c(14, 14, 16, 11, 2, 17, 13)
```

```
if (all(edades >= 18)) { # Si todas las personas son mayores de edad

  mayores_de_edad <- TRUE

  print("todos mayores de edad")

} else if (any(edades >= 18)) { # si alguna es mayor de edad

  edades_18 <- edades[edades >= 18]

  print("algún menor de edad se ha eliminado")

} else { # ninguna persona mayor de edad

  print("todas las personas son menores de edad")

}

## [1] "todas las personas son menores de edad"
```

Esta **estructura condicional puede ser vectorizada**, de forma que podemos reunir en una **sola fila un número elevado de estructuras de comparación**. Por ejemplo, vamos a definir un vector de números y vamos a comprobar si son números pares o impares (para ello, usamos el operador `%%`, que nos calcula el resto de cada número al dividirlo por una cifra).

```
1 %% 2
```

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

```
2 %% 2
```

```
## [1] 0
```

```
3 %% 2
```

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

```
5 %% 3
```

```
## [1] 2
```

Para nuestro objetivo aplicaremos la función `ifelse()`, cuyos argumentos de entrada serán la condición a evaluar, lo que sucede cuando se cumple y lo que no, que aplicará a cada elemento del vector de entrada.

```
numeros <- 1:10
ifelse((numeros %% 2) == 0, "par", "impar") # Los pares al dividir entre 2 tienen resto 0
```

```
## [1] "impar" "par"   "impar" "par"   "impar" "par"   "impar" "par"   "impar"
## [10] "par"
```

Esta función `ifelse()` es muy util para codificar variables o averiguar cuales cumplen una condición, sin necesidad de hacer un bucle que recorra todos los valores. **Recuerda: di (por lo general) no a los bucles.**

## 7.2 for/while

Aunque el 99% (porcentaje inventado, pero más o menos) de las veces los bucles pueden ser sustituidos por códigos de forma vectorial mucho más eficientes (ya hemos visto algunos ejemplos), a veces no nos quedará más remedio que usarlos

por lo que nunca viene mal conocer su estructura.

Un **bucle for{}** es una estructura que nos permite ejecutar un conjunto de órdenes un número repetido (finito y conocido) de veces: dado un conjunto de índices, el bucle irá recorriendo cada elemento de dicho conjunto, y para cada uno de ellos ejecutará lo que tenga dentro de las llaves.

```
indices <- 1:10

variable <- NULL # vector donde guardaremos los pasos del bucle

for (i in 1:10) {

    variable[i] <- i # R es silenciosos: salvo que hagamos un print dentro del bucle no nos imprimirá nada

}

variable

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

for (i in 1:length(indices)) {

    variable[i] <- i

}

variable

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

for (i in 1:length(indices)) {

    print(i^3) # imprimimos el índice al cubo

}
```

```
## [1] 1
## [1] 8
## [1] 27
## [1] 64
## [1] 125
## [1] 216
## [1] 343
## [1] 512
## [1] 729
## [1] 1000
```

Escribiendo `length(indices)`, si cambiamos la variable `indice` no necesitamos cambiar el bucle (llegará hasta el final de dicho conjunto de valores, valga lo que valga).

Aunque normalmente el conjunto que recorre el bucle suelen ser índices numéricos, **podemos recorrer cualquier tipo de objeto**.

```
dias_semana <- c("lunes", "martes", "miércoles", "jueves",
                  "viernes", "sábado", "domingo")
nombre_mayuscula <- NULL

for (dias in dias_semana) { # días recorre los días de la semana tomando sus valores

  print(toupper(dias))
}

## [1] "LUNES"
```

```
## [1] "MARTES"
## [1] "MIÉRCOLES"
## [1] "JUEVES"
## [1] "VIERNES"
## [1] "SÁBADO"
## [1] "DOMINGO"
```

**CONSEJO:** evita al máximo los bucles, suele existir una forma más eficiente de programarlo. Veamos un ejemplo muy sencillo: dado un vector de índices `idx`, queremos calcular su cuadrado y guardarla. Vamos a comparar como sería con un sencillo bucle y de forma vectorial, repitiéndolo 1000 veces para sacar tiempos medios, haciendo uso del paquete `{microbenchmark}`.

```
idx <- 1:1e4
x <- y <- rep(0, length(idx))
microbenchmark::microbenchmark(x <- idx^2,
                                for (i in idx) { y[i] <- idx[i]^2 },
                                times = 1e3)

## Unit: microseconds
##                                              expr      min       lq      mean     median      max
##                               x <- idx^2  25.959  70.395 109.1929   88.813
## for (i in idx) { y[i] <- idx[i]^2 } 2865.757 3728.834 7190.7898 4956.790
##                               uq      max  neval cld
##      119.357 2996.265  1000    a
```

```
## 6604.146 305836.035 1000 b
```

Una tarea tan sencilla, **programada en un bucle**, tarda 40 veces más que hacerlo **de forma vectorial** (elevando cada elemento al cuadrado, iterando internamente, sin necesidad de implementar un bucle).

Otra manera de diseñar un bucle es con la **estructura while{}**, que ejecutará el **bucle un número de veces a priori desconocido** hasta que la condición impuesta deje de ser TRUE.

```
max_ciclos <- 10  
ciclos <- 1  
  
# Mientras el número de ciclos sea inferior 10, imprime  
while(ciclos <= max_ciclos) {  
  
  print(paste("Todavía no, vamos por el ciclo ", ciclos)) # Pegamos la frase al número de ciclos  
  ciclos <- ciclos + 1  
  
}  
  
## [1] "Todavía no, vamos por el ciclo 1"  
## [1] "Todavía no, vamos por el ciclo 2"  
## [1] "Todavía no, vamos por el ciclo 3"  
## [1] "Todavía no, vamos por el ciclo 4"  
## [1] "Todavía no, vamos por el ciclo 5"
```

```
## [1] "Todavía no, vamos por el ciclo 6"  
## [1] "Todavía no, vamos por el ciclo 7"  
## [1] "Todavía no, vamos por el ciclo 8"  
## [1] "Todavía no, vamos por el ciclo 9"  
## [1] "Todavía no, vamos por el ciclo 10"
```

¿Y qué sucede cuando la condición nunca llega a ser FALSE? Compruébalo tú mismo/a.

```
while (1 > 0) { # Nunca va a dejar de ser cierto  
  
  print("Presiona ESC para salir del bucle")  
  
}
```

### 7.2.1 BREAK/NEXT

En R tenemos dos comandos reservados para poder **abortar un bucle** o **avanzar forzosamente un bucle**: dichas palabras son `break` y `next`. La primera nos habilita para **parar un bucle** aunque no haya llegado al final de su conjunto de índices a recorrer (o se siga cumpliendo la condición del `while{}`).

```
for(i in 1:10) {  
  if (i == 7) {  
  
    break # si i es 7, el bucle frena aquí (nunca llegará a imprimir el 7 ni los sucesivos)  
  
  }  
}
```

```
print(i)  
}
```

```
## [1] 1  
## [1] 2  
## [1] 3  
## [1] 4  
## [1] 5  
## [1] 6
```

Mientras que la segunda **obliga al bucle a avanzar a la siguiente interacción**, abortando la iteración actual en la que se encuentra.

```
for(i in 1:10) {  
  if (i == 7) {
```

*next # si i es 7, la iteración frenará aquí y pasará a la siguiente por lo que imprimirá 8, 9, 10*

```
  }  
  print(i)  
}
```

```
## [1] 1  
## [1] 2  
## [1] 3  
## [1] 4  
## [1] 5
```

```
## [1] 6  
## [1] 8  
## [1] 9  
## [1] 10
```

### 7.2.2 REPEAT

Aunque es una opción muy poco usada, existe una estructura de control llamada `repeat{}` que nos **ejecuta un bucle de forma infinita** hasta que le ordenemos parar con un `break`.

```
conteo <- 0  
repeat {  
  
  conteo <- conteo + 1  
  if (conteo >= 100) { break }  
  
}  
conteo  
  
## [1] 100
```

## 7.3 Consejos y tips

Cuidado con los bucles infinitos

Las estructuras `while{}` y `repeat{}` son de las menos usadas por su peligrosidad, ya que si no incluimos un `break` o la condición nunca llega a ser `TRUE`, el bucle

seguirá ejecutándose de forma infinita y solo podrá ser detenido abortando la ejecución con la tecla **ESC**.

Código limpio: minimizando estructuras de control en el código

Puedes minimizar las estructuras de control pulsando en la flecha que aparece a la izquierda de ellas.

## **Part III**

# **Listas y funciones**



# Chapter 8

## Tipos de datos III: listas/factores

Veamos algún tipo de dato muy particular más allá de vectores, matrices y `data.frames`.

### 8.1 Listas

Probablemente las **listas** sea uno de los tipos de datos más importantes en R ya que permiten almacenar **colecciones de variables de diferente tipo** (**ya lo hacían los `data.frames`**) pero también de diferente longitud, y con estructuras totalmente heterogéneas, de ahí que sea el formato de salida de muchísimas funciones de R que te devuelven a la vez un cadena de texto, un vector de números o una tabla, todo guardado en la misma variable (incluso una lista puede tener dentro a su vez otra lista).

Vamos a crear nuestra primera lista.

```

# Fecha de nacimiento

fecha_nacimiento <- as.Date("1989-09-10")

# Notas de asignaturas en primer y segundo parcial

notas <- data.frame("biología" = c(5, 7), "física" = c(4, 5),
                     "matemáticas" = c(8, 9.5))

# Nombre a las filas

row.names(notas) <- c("primer_parcial", "segundo_parcial")

# Números de teléfono

tlf <- c("914719567", "617920765", "716505013")

# Nombres

padres <- c("Juan", "Julia")

# Guardamos TODO en una lista (con nombres de cada elemento)

datos <- list("nacimiento" = fecha_nacimiento,
              "notas_insti" = notas, "teléfonos" = tlf,
              "nombre_padres" = padres)

datos

## $nacimiento
## [1] "1989-09-10"
##
## $notas_insti

```

```
##           biología física matemáticas
## primer_parcial      5      4      8.0
## segundo_parcial     7      5      9.5
##
## $teléfonos
## [1] "914719567" "617920765" "716505013"
##
## $nombre_padres
## [1] "Juan"  "Julia"

names(datos)

## [1] "nacimiento"    "notas_insti"    "teléfonos"      "nombre_padres"
```

Hemos creado una lista de 4 elementos: el elemento `nacimiento` (una fecha), el elemento `notas_insti` (un `data.frame`), el elemento `teléfonos` (un vector de números) y `nombre_padres` (un vector de texto).

Una lista es una variable que en un primer nivel solo tiene una dimensión por lo que si quieras saber cuantos elementos tiene debes ejecutar la función `length()`.

```
dim(datos) # devolverá NULL al no tener dos dimensiones
```

```
## NULL
```

```
length(datos)
```

```
## [1] 4
```

```
class(datos) # de tipo lista
```

```
## [1] "list"
```

Para acceder a un elemento de la lista tenemos dos opciones:

- Acceder por índice: con el operador `[[i]]` accedemos al elemento i-ésimo de la lista.
- Acceder por nombre: con el operador `$nombre_elemento` accedemos al elemento cuyo nombre sea `nombre_elemento`.

```
datos[[1]]
```

```
## [1] "1989-09-10"
```

```
datos$nacimiento
```

```
## [1] "1989-09-10"
```

```
datos[[2]]
```

```
##           biología física matemáticas
```

```
## primer_parcial      5     4      8.0
```

```
## segundo_parcial     7     5      9.5
```

```
datos$notas_insti
```

```
##           biología física matemáticas
```

```
## primer_parcial      5     4      8.0
```

```
## segundo_parcial     7     5      9.5
```

Si queremos acceder a varios elementos a la vez de la lista deberemos usar el op-

erador [].

```
datos[1:2]

## $nacimiento
## [1] "1989-09-10"

##
## $notas_insti
##                 biología física matemáticas
## primer_parcial      5      4      8.0
## segundo_parcial     7      5      9.5
```

Como hemos comentado, también podemos **aplicar la recursividad** y hacer **listas con otras listas dentro**, de forma que para acceder a cada nivel deberemos usar el operador [[]].

```
lista_de_listas <- list("lista_1" = datos[3:4], "lista_2" = datos[1:2])
names(lista_de_listas) # Nombres de los elementos del primer nivel

## [1] "lista_1" "lista_2"

names(lista_de_listas[[1]]) # Nombres de los elementos guardados en el primer elemento, que es a s

## [1] "teléfonos"      "nombre_padres"

lista_de_listas[[1]][[1]] # Elemento 1 de la lista guardada como elemento 1 de la lista superior

## [1] "914719567" "617920765" "716505013"
```

Un ejemplo de la utilidad de las listas la tenemos en los archivos R.Data de vacunas que hemos cargado en nuestro script. Con names(panel\_vacunas) podemos ver

que elementos contiene en el primer nivel.

```
names(panel_vacunas)
```

```
## [1] "AN"   "AR"   "AS"   "IB"   "CN"   "CB"   "CL"   "CM"   "CT"   "VC"
## [11] "EX"   "GA"   "RI"   "MD"   "MC"   "NC"   "PV"   "CE"   "ML"   "FFAA"
## [21] "ES"
```

Cada elemento de la lista es un `data.frame` de una comunidad autonóma, que a su vez contiene una serie de variables (columnas) para cada una de las fechas (filas): **¡nos permite guardar «datos tridimensionales»!**

```
class(panel_vacunas$ES)
```

```
## [1] "data.frame"

names(panel_vacunas$ES)

## [1] "fechas"
## [2] "ISO"
## [3] "poblacion"
## [4] "porc_pobl_total"
## [5] "poblacion_mayor_16a"
## [6] "porc_pobl_total_mayor_16a"
## [7] "dosis_entrega_pfizer"
## [8] "dosis_entrega_astra"
## [9] "dosis_entrega_moderna"
## [10] "dosis_entrega_janssen"
## [11] "dosis_entrega"
```

```
## [12] "dosis_entrega_100hab"
## [13] "porc_entregadas_sobre_total"
## [14] "dosis_diarias_entrega_pfizer"
## [15] "dosis_diarias_entrega_astra"
## [16] "dosis_diarias_entrega_moderna"
## [17] "dosis_diarias_entrega"
## [18] "dosis_7D_entrega_pfizer"
## [19] "dosis_7D_entrega_astra"
## [20] "dosis_7D_entrega_moderna"
## [21] "dosis_7D_entrega"
## [22] "dosis_7D_entrega_100hab"
## [23] "dosis_admin"
## [24] "dosis_primera"
## [25] "dosis_pauta_completa"
## [26] "dosis_admin_100hab"
## [27] "porc_admin_sobre_ccaa"
## [28] "porc_admin_vs_total"
## [29] "dosis_diarias_admin"
## [30] "dosis_diarias_admin_100hab"
## [31] "crec_diario_dosis_admin"
## [32] "dosis_diarias_primera"
## [33] "dosis_diarias_segunda"
## [34] "dosis_7D_admin"
## [35] "dosis_7D_admin_100hab"
## [36] "crec_7D_dosis_admin"
```

```
## [37] "porc_admin_vs_total_7D"
## [38] "personas_vacunadas"
## [39] "personas_pauta_completa"
## [40] "personas_1dosis"
## [41] "porc_personas_vacunadas"
## [42] "porc_personas_pauta_completa"
## [43] "porc_personas_vacunadas_16a"
## [44] "porc_personas_pauta_completa_16a"
## [45] "personas_vacunadas_diarias"
## [46] "personas_pauta_completa_diarias"
## [47] "porc_personas_vacunadas_diarias"
## [48] "porc_personas_pauta_completa_diarias"
## [49] "personas_vacunadas_7D"
## [50] "personas_pauta_completa_7D"
## [51] "porc_personas_vacunadas_7D"
## [52] "porc_personas_pauta_completa_7D"
## [53] "porc_personas_vacunadas_16a_7D"
## [54] "porc_personas_pauta_completa_16a_7D"
## [55] "crec_diario_personas_vacunadas"
## [56] "crec_7D_personas_vacunadas"
## [57] "crec_diario_personas_pauta_completa"
## [58] "crec_7D_personas_pauta_completa"
## [59] "desv_porc_admin_vs_total"
## [60] "desv_dosis_entrega"
## [61] "desv_porc_personas_vacunadas"
```

```
## [62] "desv_porc_personas_pauta_completa"  
## [63] "fecha_30vacunados ritmo7D"  
## [64] "fecha_50vacunados ritmo7D"  
## [65] "fecha_70vacunados ritmo7D"  
## [66] "fecha_30inmunizados ritmo7D"  
## [67] "fecha_50inmunizados ritmo7D"  
## [68] "fecha_70inmunizados ritmo7D"  
## [69] "fecha_30inmunizados_16a ritmo7D"  
## [70] "fecha_50inmunizados_16a ritmo7D"  
## [71] "fecha_70inmunizados_16a ritmo7D"  
  
head(panel_vacunas$ES[, 1:5])  
  
##      fechas ISO poblacion porc_pobl_total poblacion_mayor_16a  
## 1 2021-01-05 ES 47450795          100        40129822  
## 2 2021-01-06 ES 47450795          100        40129822  
## 21 2021-01-07 ES 47450795          100        40129822  
## 3 2021-01-08 ES 47450795          100        40129822  
## 11 2021-01-09 ES 47450795          100        40129822  
## 12 2021-01-10 ES 47450795          100        40129822
```

El acceso lo podemos realizar por orden que ocupa en la lista pero también de forma intuitiva con \$ y el código ISO de la comunidad autónoma. Lo mismo podemos hacer con el panel de fechas, donde ahora cada elemento de la lista es una fecha, y en cada elemento de ella, está guardada la información de cada variable (columna) y cada comunidad (fila).

```
names(panel_vacunas_fecha)

## [1] "2021-01-05" "2021-01-06" "2021-01-07" "2021-01-08" "2021-
01-09"
## [6] "2021-01-10" "2021-01-11" "2021-01-12" "2021-01-13" "2021-
01-14"
## [11] "2021-01-15" "2021-01-16" "2021-01-17" "2021-01-18" "2021-
01-19"
## [16] "2021-01-20" "2021-01-21" "2021-01-22" "2021-01-23" "2021-
01-24"
## [21] "2021-01-25" "2021-01-26" "2021-01-27" "2021-01-28" "2021-
01-29"
## [26] "2021-01-30" "2021-01-31" "2021-02-01" "2021-02-02" "2021-
02-03"
## [31] "2021-02-04" "2021-02-05" "2021-02-06" "2021-02-07" "2021-
02-08"
## [36] "2021-02-09" "2021-02-10" "2021-02-11" "2021-02-12" "2021-
02-13"
## [41] "2021-02-14" "2021-02-15" "2021-02-16" "2021-02-17" "2021-
02-18"
## [46] "2021-02-19" "2021-02-20" "2021-02-21" "2021-02-22" "2021-
02-23"
## [51] "2021-02-24" "2021-02-25" "2021-02-26" "2021-02-27" "2021-
02-28"
## [56] "2021-03-01" "2021-03-02" "2021-03-03" "2021-03-04" "2021-
```

```
03-05"  
## [61] "2021-03-06" "2021-03-07" "2021-03-08" "2021-03-09" "2021-  
03-10"  
## [66] "2021-03-11" "2021-03-12" "2021-03-13" "2021-03-14" "2021-  
03-15"  
## [71] "2021-03-16" "2021-03-17" "2021-03-18" "2021-03-19" "2021-  
03-20"  
## [76] "2021-03-21" "2021-03-22" "2021-03-23" "2021-03-24" "2021-  
03-25"  
## [81] "2021-03-26" "2021-03-27" "2021-03-28" "2021-03-29" "2021-  
03-30"  
## [86] "2021-03-31" "2021-04-01" "2021-04-02" "2021-04-03" "2021-  
04-04"  
## [91] "2021-04-05" "2021-04-06" "2021-04-07" "2021-04-08" "2021-  
04-09"  
## [96] "2021-04-10" "2021-04-11" "2021-04-12" "2021-04-13" "2021-  
04-14"  
## [101] "2021-04-15" "2021-04-16" "2021-04-17" "2021-04-18" "2021-  
04-19"  
## [106] "2021-04-20" "2021-04-21" "2021-04-22" "2021-04-23"  
names(panel_vacunas_fechas$`2021-04-23`)  
  
## [1] "ccaa"  
## [2] "fechas"  
## [3] "ISO"
```

```
## [4] "poblacion"
## [5] "porc_pobl_total"
## [6] "poblacion_mayor_16a"
## [7] "porc_pobl_total_mayor_16a"
## [8] "dosis_entrega_pfizer"
## [9] "dosis_entrega_astra"
## [10] "dosis_entrega_moderna"
## [11] "dosis_entrega_janssen"
## [12] "dosis_entrega"
## [13] "dosis_entrega_100hab"
## [14] "porc_entregadas_sobre_total"
## [15] "dosis_diarias_entrega_pfizer"
## [16] "dosis_diarias_entrega_astra"
## [17] "dosis_diarias_entrega_moderna"
## [18] "dosis_diarias_entrega"
## [19] "dosis_7D_entrega_pfizer"
## [20] "dosis_7D_entrega_astra"
## [21] "dosis_7D_entrega_moderna"
## [22] "dosis_7D_entrega"
## [23] "dosis_7D_entrega_100hab"
## [24] "dosis_admin"
## [25] "dosis_primer"
## [26] "dosis_pauta_completa"
## [27] "dosis_admin_100hab"
## [28] "porc_admin_sobre_ccaa"
```

```
## [29] "porc_admin_vs_total"
## [30] "dosis_diarias_admin"
## [31] "dosis_diarias_admin_100hab"
## [32] "crec_diario_dosis_admin"
## [33] "dosis_diarias_primera"
## [34] "dosis_diarias_segunda"
## [35] "dosis_7D_admin"
## [36] "dosis_7D_admin_100hab"
## [37] "crec_7D_dosis_admin"
## [38] "porc_admin_vs_total_7D"
## [39] "personas_vacunadas"
## [40] "personas_pauta_completa"
## [41] "personas_1dosis"
## [42] "porc_personas_vacunadas"
## [43] "porc_personas_pauta_completa"
## [44] "porc_personas_vacunadas_16a"
## [45] "porc_personas_pauta_completa_16a"
## [46] "personas_vacunadas_diarias"
## [47] "personas_pauta_completa_diarias"
## [48] "porc_personas_vacunadas_diarias"
## [49] "porc_personas_pauta_completa_diarias"
## [50] "personas_vacunadas_7D"
## [51] "personas_pauta_completa_7D"
## [52] "porc_personas_vacunadas_7D"
## [53] "porc_personas_pauta_completa_7D"
```

```

## [54] "porc_personas_vacunadas_16a_7D"
## [55] "porc_personas_pauta_completa_16a_7D"
## [56] "crec_diario_personas_vacunadas"
## [57] "crec_7D_personas_vacunadas"
## [58] "crec_diario_personas_pauta_completa"
## [59] "crec_7D_personas_pauta_completa"
## [60] "desv_porcent_admin_vs_total"
## [61] "desv_dosis_entrega"
## [62] "desv_porcent_personas_vacunadas"
## [63] "desv_porcent_personas_pauta_completa"
## [64] "fecha_30vacunados ritmo7D"
## [65] "fecha_50vacunados ritmo7D"
## [66] "fecha_70vacunados ritmo7D"
## [67] "fecha_30inmunizados ritmo7D"
## [68] "fecha_50inmunizados ritmo7D"
## [69] "fecha_70inmunizados ritmo7D"
## [70] "fecha_30inmunizados_16a ritmo7D"
## [71] "fecha_50inmunizados_16a ritmo7D"
## [72] "fecha_70inmunizados_16a ritmo7D"

head(panel_vacunas_fecha$`2021-04-23`[, 1:7])

```

##	ccaa	fechas ISO	poblacion	porc_pobl_total	poblacion_mayor_16a	
## 77	AN	2021-04-23	AN	8464411	17.838	7062213
## 771	AR	2021-04-23	AR	1329391	2.802	1132764
## 772	AS	2021-04-23	AS	1018784	2.147	901209

```
## 773   IB 2021-04-23   IB    1171543      2.469      986279
## 774   CN 2021-04-23   CN    2175952      4.586     1871033
## 775   CB 2021-04-23   CB    582905       1.228     501384
##           porc_pobl_total_mayor_16a
## 77             17.598
## 771            2.823
## 772            2.246
## 773            2.458
## 774            4.662
## 775            1.249
```

## 8.2 Factores

Los factores son el tipo de dato que tiene R para definir variables categóricas, variables que aunque puedan ser números, en realidad representa **categorías (cat-egoría 1, 2, 3...)**. Internamente los factores se guardan como variable numéricas enteras (enumerando las categorías) pero se nos mostarán con el nombre asignada a dicha categoría. Para convertir una variable a factor basta con ejecutar la función `factor()`, que nos convierte cada valor diferente en una categoría (para ver valores diferentes de un vector, usar la función `unique()`).

```
datos <- c(1, 2, 2, 3, 1, 2, 3, 3, 1, 2, 3, 3, 1)
unique(datos)
```

```
## [1] 1 2 3
```

```
datos
```

```
## [1] 1 2 2 3 1 2 3 3 1 2 3 3 1
datos_factor <- factor(datos) # Convertimos a factor
datos_factor
```

```
## [1] 1 2 2 3 1 2 3 3 1 2 3 3 1
## Levels: 1 2 3
```

Es importante entender que un **factor es una categoría**, como rojo/blanco/negro, por lo que desde el momento en el que lo convertimos en factor, ya no podemos hacer operaciones aritméticas.

```
datos + 1
```

```
## [1] 2 3 3 4 2 3 4 4 2 3 4 4 2
datos_factor + 1

## Warning in Ops.factor(datos_factor, 1): '+' not meaningful for factors

## [1] NA NA
```

Como puedes observar, al tenerlo en factor, además de la variable en sí se nos **muestra debajo los levels, los nombres de las categorías**. Si no le indicamos que nombres queremos, nos convierte los valores a texto y lo toma como nombre de categoría. Con el **argumento labels podemos configurarlo** a nuestro gusto. Con la función **levels** podemos reasignarle nombres tras su generación.

```
datos_factor <- factor(datos, labels = paste("Categoría", sort(unique(datos)))) # damos nombre de  
datos_factor  
  
## [1] Categoría 1 Categoría 2 Categoría 2 Categoría 3 Categoría 1 Categoría 2  
## [7] Categoría 3 Categoría 3 Categoría 1 Categoría 2 Categoría 3 Categoría 3  
## [13] Categoría 1  
## Levels: Categoría 1 Categoría 2 Categoría 3  
  
levels(datos_factor) <- c("C1", "C2", "C3")  
datos_factor  
  
## [1] C1 C2 C2 C3 C1 C2 C3 C3 C1 C2 C3 C3 C1  
## Levels: C1 C2 C3
```

Aunque sirve también para variables numéricas, la función **table()** nos calcula las **frecuencias de cada una de las categorías**, las veces que se repiten en nuestro conjunto (es una forma eficiente de guardar categorías ya que solo se guardan los valores únicos y el número de veces que se repiten, así como su lugar).

```
table(datos_factor)  
  
## datos_factor  
## C1 C2 C3  
## 4 4 5
```

Una ventaja de los factores es que le podemos **indicar que considere que las categorías son ordinales**: tiene una jerarquía de orden, con el argumento `ordered = TRUE`.

```
notas <- c(7, 2, 10, 5, 7, 8, 10, 8, 2, 2, 5, 5, 5, 10) # notas de clase: tienen un orden

notas_factor <- factor(notas)

notas_factor[1] < notas_factor[2]

## Warning in Ops.factor(notas_factor[1], notas_factor[2]): '<' not meaningful for

## factors

## [1] NA

notas_factor_ordenados <- factor(notas, ordered = TRUE)

notas_factor_ordenados[1] < notas_factor_ordenados[2] # nos dice que la categoría 7 no es menor que la categoría 2

## [1] FALSE

notas_factor_ordenados

## [1] 7 2 10 5 7 8 10 8 2 2 5 5 5 10

## Levels: 2 < 5 < 7 < 8 < 10
```

Para pasar de factor a variable numérica (y poder operar con ellos), basta usar la función `as.numeric()`.

notas\_factor\_ordenados + 1

```
## Warning in Ops.ordered(notas_factor_ordenados, 1): '+' is not meaningful for
## ordered factors

## [1] NA NA

as.numeric(notas_factor_ordenados) + 1

## [1] 4 2 6 3 4 5 6 5 2 2 3 3 3 3 6
```

```
mean(as.numeric(notas_factor_ordenados))
```

```
## [1] 2.857143
```

También podemos **convertir variables continuas (o discretas) a factores indicando los rangos de las categorías** que queremos asignar con la función `cut()`.

Por ejemplo, supongamos que tenemos notas numéricas de clase y queremos asignar una nota categórica. En el argumento `breaks` debemos indicarle los cortes que queremos en los datos, teniendo `n+1` valores, siendo `n` el número de categorías. Con `right = FALSE` le vamos a indicar que los intervalos son abiertos por la derecha.

```
notas <- c(7.4, 1.1, 2.9, 10, 5.2, 7.7, 8.9, 10, 8.1, 2.6, 2.4, 5.5, 5, 5, 10, 6.3, 9.4) # notas continuas

notas_categoricas <- cut(notas, breaks = c(0, 5, 7, 9, 10, 10.1), labels = c("suspenso", "aprobado"))

notas_categoricas
```

	## [1] notable	suspenso	suspenso	mh	aprobado
## [6]	notable	notable	mh	notable	suspenso
## [11]	suspenso	aprobado	aprobado	aprobado	mh
## [16]	aprobado	sobresaliente			
## Levels:	suspenso	aprobado	notable	sobresaliente	mh

Además, la función `cut()` identifica los datos de tipo fecha, pudiendo hacer **cortes por unidades temporales**.

```
fechas <- as.Date(c("2021-04-10", "2021-03-10", "2021-01-01", "2020-01-15", "2020-09-10", "2020-09-15"))

fechas_cortes <- cut(fechas, breaks = "year")

levels(fechas_cortes) <- c("2020", "2021")
```

```
fechas_cortes
```

```
## [1] 2021 2021 2021 2020 2020 2020 2020
## Levels: 2020 2021
```

### 8.3 Fechas y horas

Como ya hemos dicho, las fechas y momentos temporales no serán meras cadenas de carácter sino que tienen clases especiales asociadas y algunas funciones especiales que pueden sernos útiles: las fechas serán de tipo `dates` mientras que las horas será de tipo `POSIXct` o `POSIXlt`. En el primer caso, las fechas serán guardadas internamente como el **número de días transcurridos desde el 1 de enero de 1970**, y las horas como **número de segundos desde el 1 de enero de 1970** (para la clase `POSIXct`) o una lista de segundos, minutos y horas (para la clase `POSIXlt`).

**¿Cómo obtener automáticamente la fecha de hoy, por ejemplo?** La función `Sys.Date()` nos devuelve directamente la fecha y hora en el momento de la ejecución de la orden.

```
fecha <- Sys.Date()
fecha

## [1] "2021-09-08"

fecha - 7 # una semana antes

## [1] "2021-09-01"
```

```
class(fecha) # de clase fecha
```

```
## [1] "Date"
```

Para **convertir una cadena de texto a fecha**, basta usar la función `as.Date()` con la fecha en formato "yyyy-mm-dd" por defecto. Si le introducimos otro tipo de formato, debemos especificárselo en un segundo argumento.

```
as.Date("2021-03-10") # formato por defecto
```

```
## [1] "2021-03-10"
```

```
as.Date("10-03-2020", "%d-%m-%Y") # con día-mes-año (4 cifras)
```

```
## [1] "2020-03-10"
```

```
as.Date("10-03-20", "%d-%m-%y") # con día-mes-año (2 cifras)
```

```
## [1] "2020-03-10"
```

```
as.Date("03-10-2020", "%m-%d-%Y") # con mes-día-año (4 cifras)
```

```
## [1] "2020-03-10"
```

```
as.Date("Octubre 21, 1995 21:24", "%B %d, %Y %H:%M") # fecha escrita
```

```
## [1] "1995-10-21"
```

Fíjate la diferencia cuando lo convertimos en fecha

```
"2021-03-10" - 1 # error
```

```
## Error in "2021-03-10" - 1: argumento no-numérico para operador binario
```

```
as.Date("2021-03-10") - 1 # día previo
```

```
## [1] "2021-03-09"
```

Aunque aparentemente parezca una cadena de texto, prueba a ejecutar `unclass(fecha)` para comprobar que tiene internamente guardado.

```
unclass(fecha)
```

```
## [1] 18878
```

```
unclass(as.Date("1969-01-01")) # justo un año antes de la referencia
```

```
## [1] -365
```

Para la **fecha con hora actual** podemos usar una función similar, la función `Sys.time()`. Al igual que la función `as.Date()` para convertir cadenas de texto en fechas, podemos hacer uso de la función `strptime()` para convertir cadenas de texto en fecha-hora.

```
fecha_hora <- Sys.time() # fecha y hora actual en formato POSIXct
```

```
class(fecha_hora)
```

```
## [1] "POSIXct" "POSIXt"
```

```
unclass(fecha_hora)
```

```
## [1] 1631098222
```

```
strptime("Octubre 21, 1995 21:24", "%B %d, %Y %H:%M") # fecha escrita
```

```
## [1] "1995-10-21 21:24:00 CET"
```

```
# fecha y hora actual en formato POSIXlt  
  
fecha_hora2 <- as.POSIXlt(Sys.time())  
  
class(fecha_hora2)  
  
## [1] "POSIXlt" "POSIXt"  
  
unclass(fecha_hora2)  
  
## $sec  
## [1] 21.53896  
  
##  
  
## $min  
## [1] 50  
  
##  
  
## $hour  
## [1] 12  
  
##  
  
## $mday  
## [1] 8  
  
##  
  
## $mon  
## [1] 8  
  
##  
  
## $year  
## [1] 121  
  
##
```

```
## $wday
## [1] 3
##
## $yday
## [1] 250
##
## $isdst
## [1] 1
##
## $zone
## [1] "CEST"
##
## $gmtoff
## [1] 7200
##
## attr(,"tzone")
## [1] ""      "CET"   "CEST"

str(unclass(fecha_hora2))

## List of 11
## $ sec    : num 21.5
## $ min    : int 50
## $ hour   : int 12
## $ mday   : int 8
## $ mon    : int 8
```

```
## $ year  : int 121
## $ wday  : int 3
## $ yday  : int 250
## $ isdst : int 1
## $ zone  : chr "CEST"
## $ gmtoff: int 7200
## - attr(*, "tzone")= chr [1:3] "" "CET" "CEST"
```

```
fecha_hora2$min # Accedemos a los minutos
```

```
## [1] 50
```

Además tenemos disponibles funciones para extraer fácilmente algunas variables temporales como el día de la semana, el mes o el cuatrimestre, con las funciones `weekdays()`, `months()`, and `quarters()`.

```
weekdays(fecha)
```

```
## [1] "miércoles"
```

```
months(fecha)
```

```
## [1] "septiembre"
```

```
quarters(fecha)
```

```
## [1] "Q3"
```

Al igual que podemos realizar operaciones aritméticas sencillas con las fechas, también podemos **realizar comparaciones**, por ejemplo, si el día actual es menor o mayor que otra fecha dada.

```
fecha_actual <- Sys.Date()
fecha_actual > as.Date("2020-04-15")

## [1] TRUE

fecha_actual < as.Date("2020-04-15")

## [1] FALSE
```

La función `difftime()` nos permite además hallar distancias entre fechas pero no solo en días (lo que sucedería si restamos 2 fechas) sino en las unidades temporales que queramos.

```
diffime(Sys.time(), as.POSIXct("2020-01-17 10:24:12 CEST"), units = "days")

## Time difference of 600.0598 days

diffime(Sys.time(), as.POSIXct("2020-01-17 10:24:12 CEST"), units = "hours")

## Time difference of 14401.44 hours

diffime(Sys.time(), as.POSIXct("2020-01-17 10:24:12 CEST"), units = "weeks")

## Time difference of 85.72283 weeks
```

## 8.4 Consejos y tips

Paquete {lubridate}

En dicho paquete tienes muchas **funcionalidades para trabajar con fechas**. Ver <https://lubridate.tidyverse.org/>.

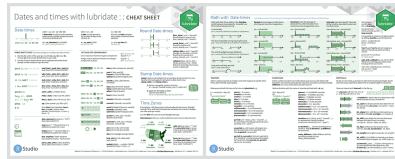


Figure 8.1: Paquete lubridate.

## Paquete {forcats}

En dicho paquete tienes muchas **funcionalidades para trabajar con factores**. Ver <https://forcats.tidyverse.org/>.

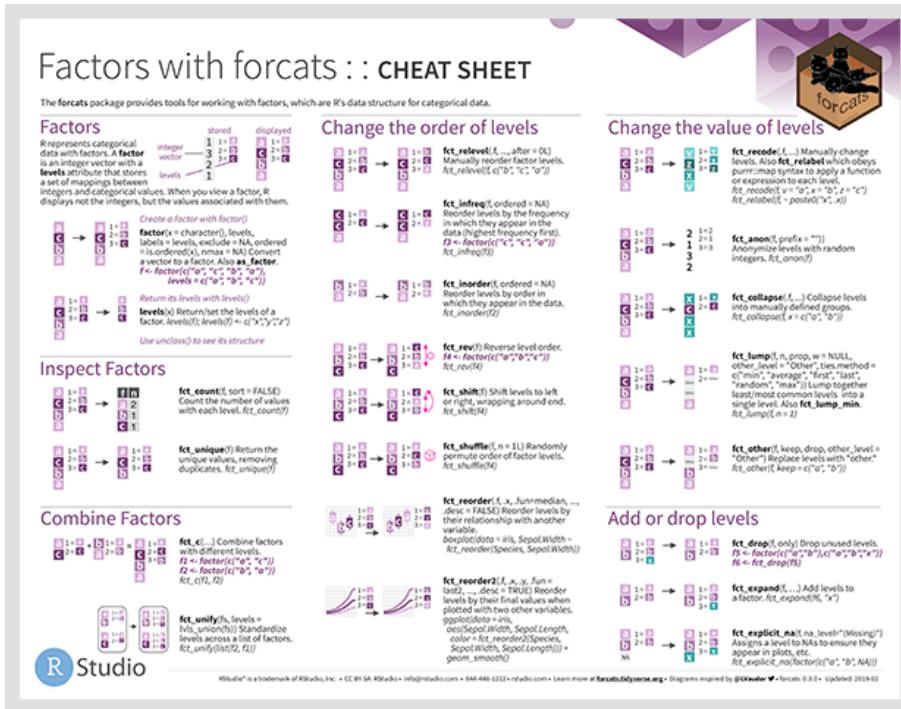


Figure 8.2: Paquete lubridate.

## Operaciones aritméticas con listas

Una **lista no se puede vectorizar de forma inmediata**, por lo cualquier operación aritmética aplicada a una lista dará error (ver más adelante la función `lapply()`).

```
datos / 2

## [1] 0.5 1.0 1.0 1.5 0.5 1.0 1.5 1.5 0.5 1.0 1.5 1.5 0.5
```

## 8.5 □ Ejercicios

Ejercicio 1: define una lista de 4 elementos de tipos distintos y accede al segundo de ellos (yo incluiré uno que sea un `data.frame` para que veas que en una lista cabe de todo).

- Solución:

```
# Ejemplo: lista con texto, numérico, lógico y un data.frame

lista_ejemplo <- list("nombre" = "Javier", "cp" = 28019,
                      "soltero" = TRUE,
                      "notas" = data.frame("mates" = c(7.5, 8, 9),
                                           "lengua" = c(10, 5, 6),
                                           "gimnasia" = c(4, 8, 6)))
lista_ejemplo
```

```
## $nombre
## [1] "Javier"
##
## $cp
## [1] 28019
```

```
##  
## $soltero  
## [1] TRUE  
##  
## $notas  
##   mates lengua gimnasia  
## 1    7.5     10      4  
## 2    8.0      5      8  
## 3    9.0      6      6
```

```
# Longitud
```

```
length(lista_ejemplo)
```

```
## [1] 4
```

```
# Accedemos al elemento dos
```

```
lista_ejemplo[[2]]
```

```
## [1] 28019
```

Ejercicio 2: define una lista de 4 elementos que contenga, en una sola variable, tu nombre, apellido, edad (como dato numérico) y si estás soltero/a.

- Solución:

```
library(lubridate)
```

```
##  
## Attaching package: 'lubridate'
```

```

## The following objects are masked from 'package:base':
##
##      date, intersect, setdiff, union

# Creamos lista: con lubridate calculamos la diferencia de años desde la fecha de nuestra nacida
lista_personal <- list("nombre" = "Javier",
                      "apellidos" = "Álvarez Liébana",
                      "edad" = time_length(interval(ymd("1989-09-10"), ymd(Sys.Date())))
                      "soltero" = TRUE)

lista_personal

## $nombre
## [1] "Javier"
##
## $apellidos
## [1] "Álvarez Liébana"
##
## $edad
## [1] 31.99452
##
## $soltero
## [1] TRUE

# Otra opción: la edad calculada con floor (quita decimales y se queda con la parte entera)
lista_personal <- list("nombre" = "Javier",
                      "apellidos" = "Álvarez Liébana",
                      "edad" = floor(time_length(interval(ymd("1989-09-10"), ymd(Sys.Date())))))

```

```
    "soltero" = TRUE)  
lista_personal  
  
## $nombre  
## [1] "Javier"  
##  
## $apellidos  
## [1] "Álvarez Liébana"  
##  
## $edad  
## [1] 31  
##  
## $soltero  
## [1] TRUE
```



## Chapter 9

# Creación de funciones

En R no solo podemos usar las funciones predeterminadas que vienen ya cargadas, o las de los paquetes que instalamos, sino que además podemos **crear nuestras propias funciones**, para automatizar tareas que vayamos a repetir a lo largo de nuestro código.

**¿Cómo crear nuestra propia función?** Veamos su sintaxis básica. Para crear una función necesitamos un `nombre_funcion` (sin espacios ni caracteres extraños), al que le asignamos la orden `function()`. Dentro de `function()` tendríamos que definir los argumentos de entrada que vamos a usar.

```
nombre_funcion <- function(argumento_1, argumento_2, ... ) {  
  
  # Código que queramos ejecutar en la función  
  código
```

```
# Salida

return(variable_salida)

}
```

- **argumento\_1, argumento\_2, ...:** serán los argumentos de entrada, los argumentos que toma la función para ejecutar el código que tiene dentro
- **código:** líneas de código que queramos que ejecute la función. **IMPORANTE:** todas las variables que definamos dentro de la función son variables locales, solo existirán dentro de la función salvo que especifiquemos lo contrario.
- **return(variable\_salida):** dentro del comando `return()` se introducirá la salida de la función, que puede ser un número, un `data.frame`, una gráfica, una matriz, o todo junto en una lista.

## 9.1 Primera función

Veamos un **ejemplo muy simple de función** para calcular el área de un rectángulo.

```
# Definición del nombre de función y argumentos de entrada

calcular_area <- function(lado_1, lado_2) {

# Cuerpo de la función

area <- lado_1 * lado_2

# Resultado que devolvemos

return(area)
```

```
}
```

**¿Cómo aplicar la función?** Con el nombre y los valores de los argumentos.

```
# Aplicación de la función con los parámetros por defecto
```

```
calcular_area(5, 3) # área de un rectángulo 5 x 3
```

```
## [1] 15
```

Imagina ahora que nos damos cuenta que el 90% de las veces el área que nos toca calcular es la de un cuadrado: R nos permite definir **argumentos en la función con valores por defecto** (tomarán dicho valor salvo que le asignemos otro). **¿Por qué no asignar lado\_2 = lado\_1 por defecto**, para ahorrar líneas de código y tiempo?

```
# Definición del nombre de función y argumentos de entrada
```

```
calcular_area <- function(lado_1, lado_2 = lado_1) {
```

```
  # Cuerpo de la función
```

```
  area <- lado_1 * lado_2
```

```
  # Resultado que devolvemos
```

```
  return(area)
```

```
}
```

```
calcular_area(lado_1 = 5) # si no indicamos nada, lado_2 = lado_1
```

```
## [1] 25
```

Compliquemos un poco la función y añadamos en la salida los valores de cada lado etiquetados como primer lado y segundo lado.

```
# Definición del nombre de función y argumentos de entrada
calcular_area <- function(lado_1, lado_2 = lado_1) {

  # Cuerpo de la función
  area <- lado_1 * lado_2

  # Resultado que devolvemos en modo lista ya que devolvemos
  # varios argumentos a la vez (podemos dar a cada elemento
  # de la lista con un nombre que nos permita identificarlo)
  return(list("area" = area, "lado_1" = lado_1, "lado_2" = lado_2))

}
```

Veamos que nos devuelve ahora

```
calcular_area(5, 3)
```

```
## $area
## [1] 15
##
## $lado_1
## [1] 5
##
## $lado_2
```

```
## [1] 3
```

Fíjate que puedes guardar la salida de forma conjunta para luego acceder a solo uno de los elementos de la lista de salida.

```
x <- calcular_area(5, 3)
```

```
x$area
```

```
## [1] 15
```

```
x$lado_1
```

```
## [1] 5
```

```
x$lado_2
```

```
## [1] 3
```

Antes nos daba igual el orden de los argumentos pero ahora no, ya que en la salida incluimos `lado_1` y `lado_2`. Es **altamente recomendable** hacer la llamada a la función indicando explícitamente los argumentos `argumento_1 = valor_1` para **mejorar la legibilidad e interpretabilidad de nuestro código** (recuerda: programa como escribirías una novela).

```
calcular_area(lado_1 = 5, lado_2 = 3)
```

```
## $area
```

```
## [1] 15
```

```
##
```

```
## $lado_1
```

```
## [1] 5
```

```
##  
## $lado_2  
## [1] 3
```

## 9.2 Segunda función

Vayamos con un **ejemplo más complejo**. Imaginemos que en nuestro código vamos a tener calcular, para cada día, el número de vacunas diarias administradas, el número de vacunas administradas en los últimos 7 días y el número de vacunadas administradas en los últimos 14 días, usando tan solo el número de vacunadas acumuladas. Para ello vamos a usar la **función `diff()` que nos calcula las diferencias de un vector dado**.

```
x <- c(1, 2, 3, 7, 10, 15, 20, 50, 100, 250, 600, 1200)  
  
diff(x) # vector con [elemento2 - elemento1, elemento3 - elemento2, elemento4 - elemento3]  
  
## [1] 1 1 4 3 5 5 30 50 150 350 600  
  
diff(x, 3) # vector con [elemento4 - elemento1, elemento5 - elemento2, elemento6 - elemento3]  
  
## [1] 6 8 12 13 40 85 230 550 1100  
  
diff(x, 7) # vector con [elemento8 - elemento1, elemento9 - elemento2, elemento10 - elemento3]  
  
## [1] 49 98 247 593 1190  
  
# Para España  
casos_diarios <- diff(panel_vacunas$ES$personas_vacunadas)  
casos_7D <- diff(panel_vacunas$ES$personas_vacunadas, 7) # diferencias a 7 días  
casos_14D <- diff(panel_vacunas$ES$personas_vacunadas, 14) # diferencias a 14 días
```

```
# Para Andalucía

casos_diarios <- diff(panel_vacunas$AN$personas_vacunadas)

casos_7D <- diff(panel_vacunas$AN$personas_vacunadas, 7) # diferencias a 7 días

casos_14D <- diff(panel_vacunas$AN$personas_vacunadas, 14) # diferencias a 14 días

# ...
```

¿Cuánto ocuparía realizar esta misma tarea para cada comunidad, y cada variable?

¿Por qué no la automatizamos?

```
datos_acumulados <- function(variable_acumulada, dias_dif = c(1, 7, 14)) {

  # Dentro de las llaves el cuerpo de la función
  acumulados_diferenciales <- NULL

  for (i in dias_dif) { # Vamos calculando tantos acumulados diferenciales como le hayamos pasado

    # A lo que teníamos, le concatenamos por columnas uno nuevo
    acumulados_diferenciales <- c(acumulados_diferenciales,
                                    rev(diff(variable_acumulada, i))[1])

  }

  # La salida de la función
  return(acumulados_diferenciales)
}
```

Como vemos, los argumentos pueden ser cualquier tipo de variable, y nos permite además **generalizar y automatizar una tarea** para que pueda ser usada incluso en algún escenario para el que no tuviéramos previsto (acumulados a ... 13 días, por ejemplo). Para que nuestra función sea realmente útil debemos intentar asignar **nombres de funciones y argumentos lo más concisos posibles y evidentes en su interpretación**.

```
datos_acumulados(panel_vacunas$ES$personas_vacunadas)

## [1] 263914 1403809 3246147

datos_acumulados(panel_vacunas$AN$personas_vacunadas)

## [1] 59583 235051 537727

datos_acumulados(panel_vacunas$ES$personas_vacunadas, dias_dif = c(1, 3, 13, 21))

## [1] 263914 816950 3073866 4779568
```

Fíjate que hemos devuelto solo el último acumulado (hemos dado la vuelta al vector resultante y nos hemos quedado con el primer elemento). Para hacer que la salida sea más interpretable, muchas de las **funciones en R tienen como salida una lista, con unos nombres asignados**.

```
datos_acumulados <- function(variable_acumulada) {

  # Datos

  dato_diario <- diff(variable_acumulada)
  dato_7D <- diff(variable_acumulada, 7)
  dato_14D <- diff(variable_acumulada, 14)
```

```
# La salida de la función como una lista, con 3 vectores

return(list("diario" = dato_diario, "7D" = dato_7D,
           "14D" = dato_14D))

}

datos_salida <- datos_acumulados(panel_vacunas$ES$personas_vacunadas)

names(datos_salida)

## [1] "diario" "7D"      "14D"

datos_salida

## $diario

## [1] 33992 33992 70653 42705 42705 42705 81950 93597 94548 92764
## [11] 42226 42225 42226 61129 53539 43950 43124 17175 17176 17175
## [21] 18624 15876 19464 19464 9682 9682 9682 3166 11265 13420
## [31] 26031 4458 4459 4458 9833 15107 33663 45331 22986 22985
## [41] 22986 36073 43687 66918 126790 42768 42769 42768 63596 106609
## [51] 140881 157476 68588 68589 68588 74417 130284 147861 208913 66537
## [61] 66537 66537 107709 105255 127181 100675 60566 60566 60566 62460
## [71] 38789 54080 26122 26123 26123 26122 41558 91262 103674 113995
## [81] 135234 135233 135234 125305 221541 231064 81155 81155 91719 91719
## [91] 227822 297306 375063 368637 172281 172282 172281 261006 313321 380661
## [101] 370506 146544 146543 146544 147228 297290 255746 263914
##
## $`7D`

## [1] 348702 408307 468863 490974 490495 490015 489536 468715 428657
```

```

## [10] 378059 328419 303368 278319 253268 210763 173100 148614 124954
## [19] 117461 109967 102474 87016 82405 76361 82928 77704 72481
## [28] 67257 73924 77766 98009 117309 135837 154363 172891 199131
## [37] 227711 260966 342425 362207 381991 401773 429296 492218 566181
## [46] 596867 622687 648507 674327 685148 708823 715803 767240 765189
## [55] 763137 761086 794378 769349 748669 640431 634460 628489 622518
## [64] 577269 510803 437702 363149 328706 294263 259819 238917 291390
## [73] 340984 428857 537968 647078 756190 839937 970216 1097606 1064766
## [82] 1010687 967173 923658 1026175 1101940 1245939 1533421 1624547 1705110
## [91] 1785672 1818856 1834871 1840469 1842338 1816601 1790862 1765125 1651347
## [100] 1635316 1510401 1403809

##
## $`14D`

## [1] 817417 836964 846922 819393 793863 768334 742804 679478 601757
## [10] 526673 453373 420829 388286 355742 297779 255505 224975 207882
## [19] 195165 182448 169731 160940 160171 174370 200237 213541 226844
## [28] 240148 273055 305477 358975 459734 498044 536354 574664 628427
## [37] 719929 827147 939292 984894 1030498 1076100 1114444 1201041 1281984
## [46] 1364107 1387876 1411644 1435413 1479526 1478172 1464472 1407671 1399649
## [55] 1391626 1383604 1371647 1280152 1186371 1003580 963166 922752 882337
## [64] 816186 802193 778686 792006 866674 941341 1016009 1078854 1261606
## [73] 1438590 1493623 1548655 1614251 1679848 1866112 2072156 2343545 2598187
## [82] 2635234 2672283 2709330 2845031 2936811 3086408 3375759 3441148 3495972
## [91] 3550797 3470203 3470187 3350870 3246147

```

### 9.3 Variables locales/globales

Hemos dicho que «lo local se queda en lo local», ¿pero qué sucede si nombramos a una variable dentro de una función que se nos ha olvidado asignar un valor dentro de la misma? Debemos ser cautos al usar funciones en R, ya que debido a la «**regla lexicográfica**», si una variable no se define dentro de la función, R buscará dicha variable en el entorno de variable.

```
x <- 1

funcion_ejemplo <- function() {

  print(x) # No devuelve nada por se, solo realiza la acción de imprimir en consola
}

funcion_ejemplo()

## [1] 1
```

Si una **variable ya está definida fuera de la función (entorno global)**, y además es usada dentro de la misma cambiando su valor, el valor de dicha variable solo cambia dentro de la función pero no en el entorno global.

```
x <- 1

funcion_ejemplo <- function() {

  x <- 2

  print(x) # lo que vale dentro

}

funcion_ejemplo() # lo que vale dentro
```

```
## [1] 2  
  
print(x) # lo que vale fuera  
  
## [1] 1
```

Si queremos que **además de cambiar localmente lo haga globalmente** deberemos usar la **doble asignación** (<<-).

```
x <- 1  
y <- 2  
funcion_ejemplo <- function() {  
  
    x <- 3 # no cambia globalmente, solo localmente  
    y <<- 0 # cambia globalmente  
    print(x)  
    print(y)  
}  
  
funcion_ejemplo() # lo que vale dentro  
  
## [1] 3  
## [1] 0  
  
x # lo que vale fuera  
  
## [1] 1  
y # lo que vale fuera  
  
## [1] 0
```

## 9.4 □ Ejercicios

Ejercicio 1: define una función propia llamada `pares` que, dados dos números `x` e `y`, nos diga si la suma de ambos es par o no.

- Solución:

```
# Definimos función
pares <- function(x, y) {

  # Sumamos
  suma <- x + y

  # Comprobamos si es par calculando el resto al dividir entre 2: si al dividir suma entre 2 el resto es 0, la suma es par
  par <- suma %% 2 == 0

  # Devolvemos la salida
  return(par)
}

# Aplicamos la función
pares(1, 3)

## [1] TRUE
```

```
pares(1, 0)
```

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

```
pares(2, 6)
```

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

```
pares(2, 7)
```

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

También se puede definir directamente como

```
# Definimos función
```

```
pares <- function(x, y) {  
  
  # Devolvemos la salida  
  return((x + y) %% 2 == 0)  
}
```

```
pares(1, 3)
```

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

```
pares(1, 0)
```

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

```
pares(2, 6)
```

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

```
pares(2, 7)
```

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

Ejercicio 2: define una función propia llamada `proximo_par` que, dados un número `x`, nos diga si es par y, en caso de no serlo, nos devuelva el próximo número que si lo sea.

- Solución:

```
# Definimos función

proximo_par <- function(x) {

  # ¿par? TRUE/FALSE
  par <- (x %% 2) == 0

  # Si es par, devolvemos el propio número (era par), sino le sumamos uno
  if (par) {

    return(list("par" = par, "proximo" = x))

  } else { # Si no es par, devolvemos el siguiente (que será par)

    return(list("par" = par, "proximo" = x + 1))

  }
}
```

```
# Devolvemos una lista de dos elementos: par (TRUE/FALSE) y proximo (si es par, el proximo)

}

# Aplicamos la función

proximo_par(7)

## $par
## [1] FALSE
##
## $proximo
## [1] 8

proximo_par(8)

## $par
## [1] TRUE
##
## $proximo
## [1] 8
```

# Chapter 10

## ¿Qué sabemos hacer?

### Hasta aquí el aperitivo

Show	7	entries	Search:	▼
variable	+	ejemplo	detalles	▼
número	x <- 1 o bien x <- c(1)	un elemento		
vector numérico	x <- c(1, 2, 3)	vector unidim. (mismo tipo)		
vector caracteres	x <- c('mi', 'nombre', 'es')	vector unidim. (mismo tipo)		
matriz	x <- matrix(1:12, nrow = 4)	vector bidimensional		
data.frame	x <- data.frame('a' = 1:3, 'b' = c('madrid', 'barcelona', 'valencia'))	colección misma longitud pero distinto tipo		
lista	x <- list('a' = 1:3, 'b' = 1:2, 'c' = c('con hijos'))	colección cualquier tipo, cualquier longitud		

Showing 1 to 6 of 6 entries      Previous  Next

Quizás creas que te queda un mundo por aprender:

- ¿Cómo manejo datos (filtrar, operar con ellos, etc)?
- ¿Cómo realizo análisis estadísticos?
- ¿Cómo visualizo datos?
- ¿Cómo generar informes con los resultados (ver ([Xie, 2015](#)))?
- ¿Cómo crear webs interactivas para la visualización y análisis de datos?

No te voy a mentir: no tendrías días de tu vida para ir investigando todos los paquetes que hay hechos en R. Pero la idea de estos primeros capítulos no era que fueses experto en R sino que vieses que **con pocas líneas de código y con cierta práctica** se puede empezar a tener herramientas para **comenzar nuestra andadura** en el análisis estadístico a través de este software. No te obsesiones con saberte todos los comandos de todos los paquetes o acabarás loco/a.



Figure 10.1: Cuando intentas aprenderte todos los paquetes.

Estos primeros capítulos suelen ser un poco «aburridos»

- **Si sabes programar**, te habrán parecido triviales (más allá de conocer la sintaxis propia de R).
- **Si no sabías programar**, han sido de repente un puñado de conceptos y cosas a recordar, que hasta que nos los vayas practicando tendrás que acudir a este manual (u otros recursos). **No te obsesiones con memorizar**: yo consulto cada día cosas que aparecen en este tutorial porque se me olvidan.

#### **Lo importante es entender, no memorizar comandos**

Pero aunque sean más aburridos estos primeros conceptos, **son necesarios para empezar a caminar**: el inicio de aprender un idioma siempre es un poco meh, pero sin las reglas básicas de gramática y un mínimo de léxico nunca podrás empezar.

## 10.1 Incursión aleatoria

Antes de acabar esta breve introducción a R merece la pena hablar de «**lo aleatorio**» y su generación en R.

### ¿Cómo se define la aleatoriedad?

Si alguna vez has interactuado con matemáticos o estadísticos (Dios te libre), seguramente es una palabra que les hayas escuchado mentar: **aleatoriedad**. Existen múltiples definiciones, y este manual tampoco pretende ser un tratado de filosofía de la ciencia, pero podemos definir la aleatoriedad de la siguiente manera:

**Aleatoriedad:** propiedad de todo proceso cuyo resultado final no se puede conocer con exactitud - a nivel individual o particular - antes de que se realice, aunque las condiciones iniciales se mantengan constantes (ejemplo: lanzar un dado).

### ¿Qué NO significa la palabra «aleatorio»?

- NO** tiene que implicar algo **caótico**.
- NO** significa que no se pueda **predecir** a nivel de conjunto.
- NO** significa que **carezca de un patrón** de comportamiento.

El ejemplo perfecto para entender las implicaciones de algo aleatorio es un dado, ya que no podemos saber con exactitud cuál será la siguiente tirada, pero tiene un patrón: si tiramos un millón de veces, aproximadamente un sexto del total de tiradas serán un 1.

En el análisis estadístico y en la programación en R nos vamos a encontrar con

múltiples situaciones en las que **lo aleatorio juega un papel importante**, y aunque este sea un tutorial muy básico e introductorio, creo que es interesante conocer **algunas formas muy sencillas de generar números aleatorios** (o...no tanto, ahora llegamos a la Sección 10.1.1).

Empecemos por lo más simple: vamos a **simular tiradas de una moneda**, asumiendo que solo tenemos dos opciones (eliminando la opción de caer de canto). Cuando tiramos una moneda es un experimento aleatorio, ya que no sabemos el resultado exacto de la siguiente tirada, pero sí sabemos que la probabilidad teórica es de 50-50, y que las únicas opciones a elegir son **cara** y **cruz**.

Una forma de ver el experimento de lanzar una moneda es pensar que tenemos una **urna con dos bolas (cara y cruz)**, y empezamos a sacar bolas de la urna (**permitiendo que al sacar una bola, se pueda devolver a la urna de nuevo**). Eso es precisamente lo que hace la función `sample()`, una función que nos seleccionará «aleatoriamente» elementos de una urna.

- `x`: los elementos distintos que tiene para elegir, que en nuestro caso serán "`cara`" y "`cruz`".
- `size`: el número de bolas que queremos sacar de la urna.
- `replace`: si tras extraer devolvemos la bola a la urna (`replace = TRUE`) o si se queda fuera (`replace = FALSE`, valor por defecto).
- `prob`: la probabilidad que tiene cada elemento en caso de no ser equiprobables (valor por defecto).

```
# Tiramos 20 veces una moneda
sample(x = c("cara", "cruz"), size = 20, replace = TRUE)
```

```
## [1] "cara" "cruz" "cara" "cara" "cara" "cara" "cruz" "cara" "cruz" "cara"
## [11] "cara" "cruz" "cara" "cara" "cara" "cruz" "cruz" "cara" "cruz" "cruz"
```

Fíjate que hemos indicado explícitamente `replace = TRUE` para decirle que aunque solo tengamos dos opciones, vamos a permitir que tras extraer una bola, la apuntemos, y la volvamos a introducir (puede salir de nuevo). ¿Qué sucede si `replace = FALSE` (su valor por defecto)?

```
# Tiramos 20 veces una moneda SIN reemplazamiento
sample(x = c("cara", "cruz"), size = 20)
```

```
## Error in sample.int(length(x), size, replace, prob): cannot take a sample larger than the population size
```

Al tener solo dos opciones, y no permitir que al extraer bolas vuelvan a la urna, tras extraer las dos únicas bolas, el proceso no puede continuar hasta los 20 lanzamientos.

Como seguramente te hayas percatado, lanzar una moneda es un experimento dicotómico, y dichos experimentos tienen una gran ventaja en programación y es que podemos escribirlo en binario: 0 para lo que llamemos fracaso (cara, por ejemplo), 1 para lo que llamemos éxito (cruz).

Generar experimentos dicotómicos de forma binario nos permite hacer cálculos sobre las tiradas de forma muy sencilla e intuitiva, ya que nos permite pasar de tener cadenas de texto a números.

```
# Tiramos 50 veces una moneda: 0 es cara, 1 es cruz
n_tiradas <- 50
tiradas <- sample(x = 0:1, size = n_tiradas, replace = TRUE)
```

```
tiradas

## [1] 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 1 1 1 0 0 0 1 0 0

## [39] 0 1 0 1 1 0 0 0 0 1 1 0

# Cantidad de cruces: sumamos los 1's

sum(tiradas)

## [1] 18

# Cantidad de caras: lo que son cruces

n_tiradas = sum(tiradas)

## [1] 32

# % de caras

cat(paste0(100 * sum(tiradas) / n_tiradas, "% de cruces"))

## 36% de cruces
```

El argumento `prob = ...` nos permite generar experimentos que sean dicotómicos pero que **no sean equiprobables**, algo similar a **lanzar una moneda trucada** (por ejemplo, 30% caras y 70% cruces). Nótese como dichas probabilidades deben ser introducidas como proporciones (divididas entre 100).

```
# Tiradas de una moneda trucada

tiradas <- sample(x = 0:1, size = n_tiradas, replace = TRUE, prob = c(0.3, 0.7))

# % de caras

cat(paste0(100 * sum(tiradas) / n_tiradas, "% de cruces"))
```

```
## 58% de cruces
```

□ Ejercicio: ¿cómo simularías 200 tiradas de un dado?

- Solución:

```
# Lo único que cambia son las opciones en la urna

sample(x = 1:6, size = 200, replace = TRUE)

## [1] 3 2 1 3 5 1 2 6 4 2 5 5 1 6 5 1 1 6 6 3 2 6 6 4 2 2 5 3 1 3 1 5 2 5 1 2 2
## [38] 5 6 1 1 5 3 5 6 3 1 3 6 4 3 4 4 3 2 5 3 5 2 2 4 4 5 2 2 2 3 4 3 5 2 6 3 4
## [75] 1 3 2 5 3 3 4 6 4 1 6 5 6 4 1 4 1 6 1 6 3 6 5 6 4 1 6 4 1 6 6 2 3 3 4 1 2
## [112] 6 6 4 6 6 5 6 6 3 3 2 5 1 2 6 2 3 5 5 2 2 2 5 5 4 3 4 3 4 1 1 5 6 4 1 4 5
## [149] 6 4 2 6 5 5 6 4 3 4 5 5 5 1 5 3 6 3 1 4 6 1 3 5 4 1 1 5 2 1 5 4 2 5 3 2 3
## [186] 6 6 3 6 5 1 5 6 1 4 4 1 4 1 5
```

### 10.1.1 Pseudoaleatoriedad

Si has hecho varias pruebas con los códigos de arriba quizás ya hayas visto que **cada vez que lanzas el código, el resultado es distinto**, algo similar a lo que sucedería si lanzas una moneda. Prueba a ejecutar este código varias veces.

```
sample(0:1, size = 20, replace = TRUE)

## [1] 0 0 1 0 0 0 0 0 1 0 1 0 0 0 0 0 1 1 0 0 1

sample(0:1, size = 20, replace = TRUE)

## [1] 1 1 1 0 0 1 0 0 1 0 0 0 0 1 1 0 0 1 1 1
```

```
sample(0:1, size = 20, replace = TRUE)
```

```
## [1] 1 0 0 1 1 0 1 1 0 0 0 1 0 1 0 1 0 0 0 0
```

### ¿Y si quisiéramos generar toda la clase la misma tirada de moneda?

Lo primero que quizás pienses es que es **imposible**, ya que al tirar una moneda en la vida real, nunca vamos a tener forma de garantizar que salgan las mismas tiradas a diferentes personas. Y **efectivamente eso sería cierto si nuestros procesos generados hubiesen sido aleatorios**, como en la vida real pero...no lo son.

Mientras esperamos a que lleguen al mainstream los ordenadores cuánticos, **TODO lo que hay en tu ordenador es determinístico**, ya que cualquier proceso se reduce a una **secuencia de bits (0's y 1's)** y un **algoritmo** (sin azar, cuyo resultado siempre será el mismo bajo las mismas condiciones iniciales). He aquí la decepción de tu vida: un ordenador «normal» NO puede generar procesos aleatorios, sino **procesos y números PSEUDOALEATORIOS**, basados en **cadenas pseudoaleatorias** generadas por un algoritmo determinístico.

Dichas secuencias aparentan ser aleatorias pero no lo son, y de hecho muchas son **periódicas**: si generamos el número suficiente de elementos de la cadena pseudoaleatoria volveremos al inicio. Muchos de los algoritmos disponibles para generar números aleatorios dependen, entre otros factores, de un **valor inicial llamada semilla** (normalmente obtenida a partir del reloj interno del ordenador): misma semilla, mismo resultado «aleatorio». Para **fijar la semilla** usaremos `set.seed()`, pasándole como argumento una secuencia de números (todos la misma).

```
set.seed(1234567)  
sample(0:1, size = 20, replace = TRUE)  
  
## [1] 0 1 1 0 1 0 0 0 1 1 0 1 0 1 0 0 1 1 1 0  
  
set.seed(1234567)  
sample(0:1, size = 20, replace = TRUE)  
  
## [1] 0 1 1 0 1 0 0 0 1 1 0 1 0 1 0 0 1 1 1 0  
  
set.seed(1234567)  
sample(0:1, size = 20, replace = TRUE)  
  
## [1] 0 1 1 0 1 0 0 0 1 1 0 1 0 1 0 0 1 1 1 0
```

**Siempre la misma tirada si la semilla inicial es la misma** ya que las cadenas pseudoaleatorias que usa el ordenador para simular nuestras tiradas son idénticas.

## 10.2 Recursos

Ahora que ya sabes lo básico para poder empezar a trabajar en un entorno amigable, aunque la idea es que este manual tenga más capítulos (¿los tiene?) para seguir avanzando, por si se me olvida, te dejo una **lista de recursos útiles** para que puedas ir viendo el abanico de opciones que tienes

- **Código de este manual:** este manual está programado en sí mismo en R y los códigos pueden ser consultados libremente en el repositorio de GitHub (hablaremos más adelante de como gestionar versiones de nuestro código en dicha plataforma).

- [Big Book of R: recopilatorio de tutoriales de R](#) en distintos campos.
- [Incursión a los modelos de regresión en R](#): manual sobre el uso de distintos modelos predictivos basados los Modelos Lineales Generalizados (GLM).
- **Paquete para aprender R**: el paquete `{swirl}` permite ir aprendiendo de forma sencilla (con preguntas tipo test) algunos conceptos básicos de R (muchos de ellos vistos en este manual). Puedes consultar la documentación en su [página web](#)
- **Manejo de datos**: probablemente el conjunto de herramientas más usadas en R sean los paquetes agrupados en `{tidyverse}` (y que veremos en capítulos sucesivos si los hubiese), un [conjunto de paquetes integrados para un manejo intuitivo de los datos](#), tanto en su preprocesamiento, como en la generación de estadísticas y gráficas.
- [Tidyverse cookbook](#)
- [Tidyverse skills for data science](#)
- **Visualización de datos en Twitter**: una de las **fortalezas de R** es su versatilidad para la visualización de datos. Y al igual que un escritor necesita leer mucho para tomar ideas, hay dos recursos en Twitter que te recomiendo encarecidamente:
  - El **hashtag #TidyTuesday** es una etiqueta en la que cada semana se plantea el reto de proponer la mejor visualización para un conjunto de datos dado, donde no solo puedes participar con la comunidad sino [ver las visualizaciones de otros usuarios de R](#).

- Además he elaborado una [lista de Twitter](#) de usuarios que se dedican a la visualización de datos.
- **Paquetes para la visualización de datos:** los paquetes `{ggplot2}` y `{plotly}` son probablemente los paquetes por excelencia en R para la visualización de datos. El **primero es uno de los paquetes más potentes de R, dentro del entorno `{tidyverse}`**, que no solo nos permite crear gráficos muy limpios y elegantes con pocas líneas sino que su sintaxis es muy intuitiva respecto a los datos visualizados. La idea inicial es poder aprender algo de `{ggplot2}` en futuros capítulos pero puedes empezar si quieras echando un vistazo a su [web oficial](#). También te recomiendo el [tutorial de Cédric Scherer](#). El segundo paquete, `{plotly}` puede llegar a tener las mismas funcionalidades pero su programación es más tediosa. Su principal ventaja es que **genera gráficos interactivos: gráficos HTML (como si fuera una página web)** (no una imagen estática) que permite al usuario interactuar con los datos pasando el ratón (incluso permite crear menús). Un ejemplo es la [web de visualización covid de Asturias](#) que elaboré durante la pandemia para el Gobierno de Asturias (la web en sí está elaborada con `{shiny}`, un paquete de R que [permite crear aplicaciones web](#)).
- **Generación de informes desde R:** el paquete `{rmarkdown}` permite [generar directamente informes](#) que mezclen texto, fórmulas matemáticas, gráficas y código R (como este mismo manual), sin necesidad de importarlos a otras herramientas de Office.
- **Comunidad de R hispano:** tenemos un [grupo de Discord](#) y [grupo de Telegrama](#)

gram varios usuarios de R en España para compartir recursos.

# Bibliography

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Ziemann, M., Eren, Y., and El-Osta, A. (2016). Gene name errors are widespread in the scientific literature. *Genome Biology*, 17(177).