R para principiantes

Juan Bosco Mendoza Veqa

R para principiantes

Juan Bosco Mendoza Vega

Prefacio

Juan Bosco Mendoza Veqa

Este libro está dirigído a personas que nunca han usado R o ningún otro lenguaje de programación. No es necesario conocimiento previo de estadística.

El propósito es que el lector:

- conozca las características principales de R
- se familiarice con los tipos y estructuras de datos usados en R
- pueda definir sus propias funciones

Este libro está enfocado a aprender el uso de R como un lenguaje de programación.

Bibliografía básica de referencia

- Diez, D., Barr, C., Çetinkaya-Rundel, M. (2015). OpenIntro Statistics, segunda edición. OpenIntro. https://www.openintro.org/stat/textbook.php?stat_book=os
- Grolemund y Wickham. (2017). R for Data Science. O'Reilly. http://r4ds.had.co.nz/
- Navarro, D. (2015). Learning statistics with R: A tutorial for psychology students and other beginners. University of Adelaide. http://www.fon.hum.uva.nl/paul/lot2015/Navarro2014.pdf
- Peng, R. D. (2016). R Programming for Data Science. Leanpub. https://leanpub.com/rprogramming
- RStudio Cheat Sheets. https://www.rstudio.com/resources/cheatsheets/
- Wickham, H. (2014) Advanced R. O'Reilly. http://adv-r.had.co.nz/
- Wickham, H. (2014) Tidy Data. Journal of Statistical Software. https://www.jstatsoft.org/article/view/v059i10/v59i10.pdf

1 Introducción: ¿Qué es R y para qué es usado?

R es un lenguaje de programación y entorno computacional dedicado a la estadística.

Decimos que es un lenguaje de programación porque nos permite dar instrucciones, usando código, a nuestros equipos de cómputo para que realicen tareas específicas (además de que es Turing Completo, pero profundizaremos en ello); para ello sólo necesitamos un intérprete para este código y es a esto a lo que llamamos un entorno computacional.

Cuando instalamos R en nuestra computadora en realidad lo que estamos instalando es el entorno computacional, y para que podamos hacer algo en ese entorno necesitamos conocer la manera de escribir instrucciones que el software pueda interpretar y ejecutar. Eso es lo que aprenderemos a hacer en este curso.

R es diferente a otros lenguajes de programación que por lo general están diseñados para realizar muchas tareas diferentes; esto es porque fue creado con el único propósito de hacer estadística. Esta característica es la razón de que R sea un lenguaje de programación peculiar, que puede resultar absurdo en algunos sentidos para personas con experiencia en otros lenguajes, pero también es la razón por la que R es una herramienta muy poderosa para el trabajo en estadística, puesto que funciona de la manera que una persona especializada en esta disciplina desearía que lo hiciera.

Para entender mejor estas peculiaridades, nos conviene conocer un poco de los orígenes de este lenguaje de programación.

1.1 Un poco de historia

R tiene sus orígenes en S, un lenguaje de programación creado en los Laboratorios Bell de Estados Unidos. Sí, los mismos laboratorios que

inventaron el transistor, el láser, el sistema operativo Unix y algunas otras cosas más.

Dado que S y sus estándares son propiedad de los Laboratorios Bell, lo cual restringe su uso, Ross Ihaka y Robert Gentleman, de la Universidad de Auckland en Nueva Zelanda, decidieron crear una implementación abierta y gratuita de S. Este trabajo, que culminaría en la creación de R inició en 1992, teniendo una versión inicial del lenguaje en 1995 y en el 2000 una versión final estable.

R hereda muchas características de S, por lo que puedes correr código de este lenguaje usando R sin mayor problema. Para lograr esto, en R frecuentemente existe más de una manera de realizar tareas comunes, una compatible con S y otra diseñada específicamente para R. Lo anterior tiene como resultado inconsistencias, sintaxis poco intuitiva y abundante frustración de cabeza para las personas que quieren aprender R.

En el presente, el mantenimiento y desarrollo de R es realizado por el R Development Core Team, un equipo de especialistas en ciencias computacionales y estadística provenientes de diferentes instituciones y lugares alrededor del mundo. La versión de R mantenida por este equipo es conocida como “base” y como su nombre indica, es sobre aquella que se crean otras implementaciones de R así como los paquetes que expanden su funcionalidad.

Para lograr que R sea usado sin restricciones es distribuido de manera gratuita, a través de la Licencia Pública General de GNU, por lo que es software libre y de código abierto. Si lo deseas, puedes examinar y estudiar el código que hace que R funcione o puedes crear versiones propias de R que se ajusten a tus necesidades particulares. Esta licencia también te permite usar R para los fines que desees, sin limitaciones, no importando si personales, académicos o comerciales.

En la actualidad, el desarrollo de este lenguaje de programación se mantiene activa. La versión más reciente de R al momento de escribir este documento es la 3.4.2 “Short Summer” fue publicada en septiembre del 2017 y diariamente son publicados nuevos paquetes y sus respectivas actualizaciones.

1.2 ¿Quién usa R?

R es un lenguaje relativamente joven pero que ha experimentado un crecimiento acelerado en su adopción durante los últimos 10 años.

En septiembre de 2017, de acuerdo al TIOBE programming community index (2017), que es uno de los índices de más prestigio en el mundo en relación popularidad en el uso de lenguajes de programación, R era el lenguaje número 11 en popularidad, después de haber sido el lenguaje número 18 en el 2016. Esto es sobresaliente si consideramos que R es un lenguaje dedicado únicamente a la estadística, mientras que lenguajes como Python (número 5 en 2017) o Java (número 1) son lenguajes que pueden ser usados para todo tipo de tareas, desde crear sitios web hasta programar robots.

La adopción de R se debe en gran medida a que permite responder preguntas mediante el uso de datos de forma efectiva, y como es un lenguaje abierto y gratuito, se facilita compartir código, crear herramientas para solucionar problemas comunes y que todo tipo de personas interesadas en análisis estadísticos puedan participar y contribuir al desarrollo y uso de R, no sólo aquellas que tengan acceso a licencias de software cerrado.

Incluso compañías e instituciones que no tendrían ninguna dificultad para financiar el costo de licencias de software cerrado utilizan R.

R, por citar un ejemplo, es usado por Facebook para analizar la manera en que sus usuarios interactúan con sus muros de publicaciones para así determinar qué contenido mostrarles. Esta es una tarea muy importante en Facebook, pues las interacciones de los usuarios con publicidad y contenido pagado son la principal fuente de ingreso de esta compañía. Además de que su división de recursos humanos emplea esta herramienta para estudiar las interacciones entre sus trabajadores.

Google usa R para analizar la efectividad las campañas de publicidad implementadas en sus servicios, por ejemplo, los anuncios pagados que te aparecen cuando “googleas” algo. Nuevamente, esta es la principal fuente

de ingresos de esta compañía. R También es usado para hacer predicciones económicas y otras actividades.

Microsoft adquirió y ahora desarrolla una versión propia de R llamada OpenR, que ha hecho disponible para uso general del público. OpenR es empleada para realizar todo tipo de análisis estadísticos, por ejemplo, para empatar a jugadores en la plataforma de videojuegos XBOX Live (así que puedes culpar a R cuando te tocan partidas contra jugadores mucho más hábiles que tú).

Otras compañías que usan R de modo cotidiano son American Express, IBM, Ford, Citibank, HP y Roche, entre muchas más (Bhalla, 2016; Level, 2017; Microsoft, 2014).

Lo anterior ilustra algunas de las aplicaciones específicas de este lenguaje y de manera general podemos decir que R es usado para procesar, analizar, modelar y comunicar datos.

Aunque R está diseñado para análisis estadístico, con el paso del tiempo los usuarios de este lenguaje han creado extensiones a R, llamadas paquetes, que han ampliado su funcionalidad. En la actualidad es posible realizar en R minería de textos, procesamiento de imagen, visualizaciones interactivas de datos y procesamiento de Big Data, entre muchas otras cosas.

Así que, empecemos a usar R.

Referencias

- Level (2017). How Big Companies Are Using R for Data Analysis. Recuperado en septiembre de 2017 de: http://www.northeastern.edu/levelblog/2017/05/31/big-companies-using-r-data-analysis/
- Microsoft (2014). Companies using R in 2014. Recuperado en septiembre de 2017 de: http://blog.revolutionanalytics.com/2014/05/companies-using-r-in-2014.html
- Bhalla, D. (2016) Companies using R. Recuperado en septiembre de 2017 de: http://www.listendata.com/2016/12/companies-using-r.html

- R FAQ. Recuperado en Septiembre de 2017 de: https://cran.r-project.org/doc/FAQ/R-FAQ.html#What-is-R_003f
- TIOBE Index for September 2017. Recuperado en Septiembre de 2017 de: https://www.tiobe.com/tiobe-index/
- Adesanya, T. (2017). A Gentler Introduction to Programming. Recuperado en Septiembre de 2017 de: https://medium.freecodecamp.org/a-gentler-introduction-to-programming-707453a79ee8

2 Instalación

La manera de instalar R cambia dependiendo del sistema operativo utilices pero todas tienen en común el uso de **CRAN**.

CRAN es el *The Comprehensive R Archive Network*, una red en la que se archivan todas las versiones de R *base*, así como todos los paquetes para R que han pasado por un proceso de revisión riguroso, realizado por el *CRAN Team*, que se encarga de asegurar su correcto funcionamiento.

CRAN es una red porque existen copias de su contenido en diferentes servidores alrededor del mundo, los cuales se actualizan diariamente. De este modo, no importa de qué servidor de CRAN descargues R o algún paquete, lo que vas a obtener será la versión más reciente de ese recurso, que es igual a la disponible en todos los demás servidores.

Como veremos más adelante, cuando descargamos un paquete de R, lo estamos haciendo desde CRAN, a menos que indiquemos otra cosa.

El sitio oficial de CRAN, en el que encontrarás más información sobre este repositorio es el siguiente:

- https://cran.r-project.org/

2.1 Windows

Para instalar R en Windows, la forma más simple es descargar la versión más reciente de R *base* desde el siguiente enlace de CRAN:

- https://cran.r-project.org/bin/windows/base/

El archivo que necesitamos tiene la extensión **.exe** (por ejemplo *R-3.5.1-win.exe*). Una vez descargado, lo ejecutamos como cualquier instalable.

Después de la instalación, estamos listos para usar R.

2.2 OSX

Para instalar R en OSX, se sigue un procedimiento similar que en Windows. Necesitamos descargar los archivos binarios de R base desde CRAN y ejecutarlos.

- https://cran.r-project.org/bin/macosx/

Al concluir la instalación, podremos usar R, incluso llamándolo directamente desde la consola.

2.3 Linux

En Linux, como suele ser el caso para casi todo, hay una manera fácil y una difícil de instalar R.

La manera fácil depende de la presencia de R en los repositorios de la distribución de Linux que estés usando. Si R se encuentra en los repositorios de tu distribución, sólo es necesario usar el gestor de paquetes de tu preferencia para instalarlo, como cualquier otro *software*.

Si R no se encuentra en los repositorios, debes agregar una entrada a tu lista de fuentes de software. Esta entrada depende de tu distribución.

También tienes la opción de puedes compilar R directamente desde archivos fuente.

Para todas las opciones anteriores, los detalles de instalación se se encuentran en el siguiente enlace:

- https://cran.r-project.org/bin/linux/

Si estás usando Linux no te debería ser difícil seguir las instrucciones presentadas.

2.4 RStudio - un IDE para R

Aunque podemos usar R directamente, es recomendable instalar y usar un entorno integrado de desarrollo (*IDE*, por sus siglas en inglés).

Podemos utilizar R ejecutando nuestro código directamente desde documentos de texto plano, pero esta es una manera poco efectiva de trabajar, especialmente en proyectos complejos.

Un IDE nos proporciona herramientas para escribir y revisar nuestro código, administrar los archivos que estamos usando, gestionar nuestro entorno de trabajo y algunas otras herramientas de productividad. Tareas que serían difíciles o tediosas de realizar de otro modo, son fáciles a través de un IDE.

Hay varias opciones de IDE para R, y entre ellas mi preferido es RStudio. Este entorno, además de incorporar las funciones esenciales de una IDE, es desarrollado por un equipo que ha contribuido de manera significativa para lograr que R sea lenguaje de programación más accesible, con un énfasis en la colaboración y la reproducción de los análisis.

Para instalar RStudio, es necesario con descargar y ejecutar alguno de los instaladores disponibles en su sitio oficial. Están disponibles versiones para Windows, OSX y Linux.

- https://www.rstudio.com/products/rstudio/download/

Si ya hemos instalado R en nuestro equipo, RStudio lo detectará automáticamente y podremos utilizarlo desde este entorno. Si no instalamos RStudio antes que R, no hay problema, cada vez que iniciamos este programa, verificará la instalación de R.

3 Conceptos básicos

Para trabajar con R es necesario conocer un poco del vocabulario usado en en este lenguaje de programación. Los siguientes son conceptos básicos que usaremos a lo largo de todo el libro.

3.1 La consola de R

Lo primero que nos encontramos al ejecutar R es una pantalla que nos muestra la versión de este lenguaje que estamos ejecutando y un *prompt*:

```
>_
```

Esta es la consola de R y corresponde al **entorno computacional** de este lenguaje. Es aquí donde nuestro código es interpretado.

Podemos escribir código directamente en la consola y R nos dará el resultado de lo pidamos allí mismo. Esta es la razón por la que se dice que R permite el uso interactivo, pues no es necesario compilar nuestro código para ver sus resultados.

Si estás usando RStudio, te encontrarás la consola de R en uno de los páneles de este programa.

3.2 Ejecutar, llamar, correr y devolver

Cuando hablamos de ejecutar, llamar o correr nos referimos a pedir que R realice algo, en otras palabras, estamos dando una instrucción o una *entrada*.

Cuando decimos que R nos devuelve algo, es que ha realizado algo que le hemos pedido, es decir, nos está dando una *salida*.

Por ejemplo, si escribimos los siguiente en la consola lo siguiente y damos Enter, estamos pidiendo que se ejecute esta operación: `> 1 + 1`

Y nos será devuelto su resultado: `[1] 2`

3.3 Objetos

En R, todo es un objeto. Todos los datos y estructuras de datos son objetos. Además, todos los objetos tienen un nombre para identificarlos.

La explicación de esto es un tanto compleja y se sale del alcance de este libro. Se relaciona con el paradigma de **programación orientada a objetos** y ese es todo un tema en sí mismo.

Lo importante es que recuerdes que al hablar de un objeto, estamos hablando de cualquier cosa que existe en R y que tiene un nombre.

3.4 Constantes y variables

De manera análoga al uso de estos términos en lenguaje matemático, una constante es un objeto cuyo valor no podemos cambiar, en contraste, una variable es un objeto que puede cambiar de valor.

Por ejemplo, en la siguiente expresión, π y **2** son constantes, mientras que **a** y **r** son variables.

$a = \pi r^2$

Las constantes y variables en R tienen nombres que nos permiten hacer referencia a ellas en operaciones.

Las constantes ya están establecidas por R, mientras que nosotros podemos crear variables, asignándoles valores a nombres.

En R usamos `<-` para hacer asignaciones. De este modo, podemos asignar el valor **3** a la variable **radio**

```
radio <- 3
```

Hablaremos sobre asignaciones más adelante, en el capítulo de operadores.

Es recomendable que al crear una variable usemos **nombres claros, no ambiguos y descriptivos**. Esto previene confusión y hace que nuestro código sea más fácil de comprender por otras personas o por nosotros mismos en el futuro.

Los nombres de las variables pueden incluir letras, números, puntos y guiones bajos. Deben empezar siempre con una letra o un punto y si empiezan con un punto, a este no puede seguirle un número.

Finalmente, cuando te encuentres con un renglón de código que inicia con un gato (hashtag), esto representa un comentario, es código que no se ejecutará, sólo se mostrará.

```
# Este es un comentario
```

3.5 Funciones (introducción básica)

Una función es **una serie de operaciones a la que les hemos asignados un nombre**. Las funciones aceptan **argumentos**, es decir, especificaciones sobre cómo deben funcionar.

Cuando llamamos una función, se realizan las operaciones que contiene, usando los argumentos que hemos establecido.

En R reconocemos a una función usando la notación: `nombre_de_la_función()`. Por ejemplo:

- `mean()`
- `quantile()`
- `summary()`
- `density()`
- `c()`

Al igual que con las variables, se recomienda que los nombres de las funciones sean claros, no ambiguos y descriptivos. Idealmente, el nombre de una función describe lo que hace. De hecho, es probable que adivines qué hacen casi todas funciones de la lista de arriba a partir de su nombre.

Aunque estrictamente hablando una función es un objeto, para fines de explicación, en este libro nos referiremos a ambos como si fueran cosas diferentes.

Las funciones son un tema que revisamos más adelante. Por el momento, recuerda que una función realiza operaciones y nos pide argumentos para poder llevarlas a cabo.

3.6 Documentación

Las funciones de R *base* y aquellas que forman parte de paquete tienen un archivo de documentación.

Este archivo describe qué hace la función, sus argumentos, detalles sobre las operaciones que realiza,los resultados que devuelve y ejemplos de uso.

Para obtener la documentación de una función, escribimos el `?` antes de su nombre y lo ejecutamos. También podemos usar la función `help()`, con el nombre de la función.

Los dos procedimientos siguientes son equivalentes.

```
?mean()
```

```
help("mean")
```

Si usas RStudio, la documentación de la función se mostrará en uno de los paneles de este IDE. Si estas usando R directamente, se abrirá una ventana de tu navegador de internet.

También podemos obtener la documentación de un paquete, si damos el argumento `package` a la función `help()`, con el nombre de un paquete.

Por ejemplo, la documentación del paquete **stats**, instalado por defecto en R *base*.

```
help(package = "stats")
```

3.7 Directorio de trabajo

El directorio o carpeta de trabajo es el lugar en nuestra computadora en el que se encuentran los archivos con los que estamos trabajando en R. Este es el lugar donde R buscara archivos para importarlos y al que serán exportados, a menos que indiquemos otra cosa.

Puedes encontrar cuál es tu directorio de trabajo con la función `getwd()`. Sólo tienes que escribir la función en la consola y ejecutarla.

```
getwd()
```

```
## [1] "C:/Users/JuanBosco/Documents/GitHub/r-principiantes-bookdown"
```

Se mostrará en la consola la ruta del directorio que está usando R.

Puedes cambiar el directorio de trabajo usando la función `setwd()`, dando como argumento la ruta del directorio que quieres usar.

```
setwd("C:\otro_directorio")
```

Por último, si deseas conocer el contenido de tu directorio de trabajo, puedes ejecutar. la función `list.files()`, sin argumentos, que devolverá una lista con el nombre de los archivos de tu directorio de trabajo. La función `list.dirs()`, también sin argumentos` te dará una lista de los directorios dentro del directorio de trabajo.

```
# Ver archivos
list.files()

# Ver directorios
list.dirs()
```

3.7.1 Sesión

Los objetos y funciones de R son almacenados en la memoria RAM de nuestra computadora.

Cuando ejecutamos R, ya sea directamente o a través de RStudio, estamos creando una instancia del entorno del entorno computacional de este lenguaje de programación. **cada instancia es una sesión**.

Todos los objetos y funciones creadas en una sesión, permanecen sólo en ella, no son compartidos entre sesiones, sin embargo una sesión puede tener el mismo directorio de trabajo que otra sesión.

Es posible tener más de una sesión de R activa en la misma computadora. Aunque ambas

Cuando cerramos R, también cerramos nuestra sesión. Se nos preguntará si deseamos guardar el contenido de nuestra sesión para poder volver a ella después. Esto se guarda en un archivo con extensión **.Rdata* en tu directorio de trabajo.

Para conocer los objetos y funciones que contiene nuestra sesión, usamos la función `ls()`, que nos devolverá una lista con los nombres de todo lo guardado en la sesión.

```
ls()
```

```
##  [1] "arboles"             "area_cuad"           "area_prisma"
##  [4] "banco"               "bato"                "bcancer"
##  [7] "bmi"                 "cara"                "coco_jambo"
## [10] "columna"             "columnas"            "conteo"
## [13] "correlaciones"       "crear_histograma"    "dado"
## [16] "desv_est"            "df"                  "edades"
## [19] "estatura"            "i"                   "ingreso"
## [22] "iris_csv"            "iris_excel"          "iris_txt"
## [25] "lista_recursiva"     "mat_cuant"           "mat_media"
## [28] "matriz"              "matriz_t"            "matriz2"
## [31] "matriz3"             "media"               "mi_array"
## [34] "mi_df"               "mi_lista"            "mi_lista_importado"
## [37] "mi_matriz"           "mi_vector"           "mi_vector_1"
## [40] "mi_vector_2"         "mi_vector_3"         "mi_vector_mezcla"
## [43] "mi_vector_nuevo"     "nivel"               "nombre"
## [46] "nombres"             "num"                 "numero"
## [49] "peso"                "posicion"            "promedio"
## [52] "ptab_banco"          "radio"               "resultado"
## [55] "tab_banco"           "tablas"              "tiempo_final"
## [58] "tiempo_inicial"      "trees_excel"         "trees_max"
## [61] "umbral"              "valor"               "variacion_tiempo"
## [64] "variacion_velocidad" "vector"              "vector_1"
## [67] "vector_2"            "vector_3"            "vector_4"
## [70] "vector1"             "vector2"             "velocidad_final"
## [73] "velocidad_inicial"
```

De manera más precisa, nuestra sesión es un **entorno** de trabajo y los objetos pertenecen a un entorno específico.

Los entornos son un concepto importante al hablar de lenguajes de programación, pero también son un tema que sale del alcance de este libro.

Con que recuerdes que **cada sesión de R tiene su propio entorno global**, eso será suficiente.

3.8 Paquetes

R puede ser expandido con **paquetes**. Cada paquete es una colección de funciones diseñadas para atender una tarea específica. Por ejemplo, hay paquetes para trabajo visualización geoespacial, análisis psicométricos, mineria de datos, interacción con servicios de internet y muchas otras cosas más.

Estos paquetes se encuentran alojados en **CRAN**, así que pasan por un control riguroso antes de estar disponibles para su uso generalizado.

Podemos instalar paquetes usando la función `install.packages()`, dando como argumento el nombre del paquete que deseamos instalar, entre comillas.

Por ejemplo, para instalar el paquete **readr**, corremos lo siguiente.

```
install.packages("readr")
```

Hecho esto, apareceran algunos mensajes en la consola mostrando el avance de la instalación

Una vez concluida la instalación de un paquete, podrás usar sus funciones con la función `library()`. Sólo tienes que llamar esta función usando como argument oel nombre del paquete que quieres utilizar

```
library(readr)
```

Cuando haces esto, R importa las funciones contenidas en el paquete al entorno de trabajo actual.

Es importante que tengas en mente que debes hacer una llamada a `library()` cada que inicies una sesión en R. Aunque hayas importado las funciones de un paquete con anterioridad, las sesiones de R se inician "limpias", sólo con los objetos y funciones de *base*.

Este comportamiento es para evitar problemas de compatibilidad y para propiciar buenas prácticas de colaboración.

Si importamos paquetes automáticamente y usamos sus funciones sin indicar de donde provienen, al compartir nuestro código con otras personas, estas no tendrán la información completa para entender qué estamos haciendo. R, al pedirnos que cada sesión indiquemos qué estamos importando, nos obliga a ser explícito con todo lo que estamos haciendo. Es un poco latoso, pero te acostumbras a ello.

En caso de escribir en `install.packages()` el nombre de un paquete no disponible en **CRAN**, se nos mostrará una advertencia y no se instalará nada.

```
install.packages("un_paquete_falso")
```

Los paquetes que hemos importado en nuestra sesión actual aparecen al llamar `sessionInfo()`.

También podemos ver qué paquetes tenemos ya instalados ejecutando la función `installed.packages()` sin ningún argumento. Una instalación nueva de R tiene

pocos paquetes instalados, pero esta lista puede crecer considerablemente con el tiempo.

4 Tipos de datos

En R los datos pueden ser de diferentes tipos. Los siguientes son los comunes.

Tipo	Ejemplo	Nombre en inglés
Entero	1	integer
Numérico	1.3	numeric
Complejo	1+0i	complex
Cadena de texto	"uno"	character
Factor	uno	factor
Lógico	TRUE	logical
Perdido	`NA`	NA
Vacio	`NULL`	null

Cada tipo de dato tiene características particulares que lo hace diferente a los demás.

Los datos numéricos también son llamados **dobles** o **floats (flotantes)**. Su nombre se deb a que son números de doble precisión, pues tienen una parte entera y una fraccionaria decimal, y son llamados floats debido a que se usa un punto flotante para su representación computacional. Para fines prácticos todos estos términos son sinónimos y son usados de manera mas o menos intercambiable en libros y documentación, lo mismo ocurrirá en este libro.

Los datos de tipo lógico sólo tienen dos valores posibles: `TRUE` (verdadero) y `FALSE`(falso). Este tipo de datos es esencial para trabajar con álgebra booleana.

`NA` se usa para representar datos perdidos y en R es diferente a `NULL`. `NULL` representa la ausencia de datos, es decir, que no hay nada que recuperar, mientras que `NA` representa un valor específico, de un dato que se reporta como perdido o ausente.

El tipo *character* representa cadenas de texto y es fácil reconocerlo porque un dato siempre esta rodeado de comillas, simples o dobles.

Las cadenas de texto son diferentes a los factores aunque suelen lucir igual en la consola. Un factor puede ser descrito como un dato numérico representado por una etiqueta. Cada uno de las etiquetas o valores que puedes asumir un factor se conoce como **nivel**.

Por ejemplo, podemos tener un conjunto de datos de tipo factor con dos niveles, **femenino (1)** y **masculino (2)**. Para nuestra computadora, femenino tiene un valor de 1, pero a nosotros se nos muestra la palabra *femenino*. Por su parte, las cadenas de texto no son representaciones de datos numéricos, son

Por esta razón, si intentamos coercionar un factor a tipo numérico, tendremos éxito, pero si intentamos lo mismo con una cadena de texto, se nos mostrará una advertencia y el resultado será **NA**.

Podemos usar la función **class()** para determinar el tipo de un dato.

```
class(3)
```

```
## [1] "numeric"
```

```
class("3")
```

```
## [1] "character"
```

```
class(TRUE)
```

```
## [1] "logical"
```

4.1 Coerción

En R, los datos pueden ser coercionados (forzados) para convertirlos de un tipo a otro. Cuando pedimos a R que ejecute una operación, R intentará

coercionar los datos al tipo correcto que permita realizarla. Si no lo logra, nos devuelve como resultado un error.

La coerción de tipos se realiza de los más restrictivos a los más flexibles.

Como los datos de tipo lógico sólo admiten dos valores (`TRUE` y `FALSE`), estos son los más restrictivos; mientras que los datos de cadena de texto, al admitir cualquier cantidad y combinación de caracteres, son los más flexibles.

Por ejemplo, podemos coercionar un dato de tipo entero a tipo numérico, pero no un dato de tipo cadena de texto a numérico.

Las coerciones siguen este orden:

```
logical -> integer -> numeric -> complex -> character
```

Podemos pedir a R que haga coerción usando la familia de funciones `as()`.

Función	Tipo al que hace coerción
`as.integer()`	Entero
`as.numeric()`	Numerico
`as.complex()`	Complejo
`as.character()`	Cadena de texto
`as.factor()`	Factor
`as.logical()`	Lógico
`as.null()`	`NULL`

Por ejemplo, con `as.character()`, podemos convertir un número en una cadena de texto.

```
as.character(5)
```

```
## [1] "5"
```

Estas funciones siguen las reglas de coerción de R, así que no podemos hacer forzar coerciones no permitidas.

```
as.logical("palabra")
```

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

4.2 Verificar el tipo de un dato

También podemos verificar si un dato es de un tipo específico con la familia de funciones `is()`.

Función	Tipo que verifican
`is.integer()`	Entero
`is.numeric()`	Numerico
`is.complex()`	Complejo
`is.character()`	Cadena de texto
`is.factor()`	Factor
`is.logical()`	Lógico
`is.null()`	`NULL`

Si el dato es del tipo que estamos verificando, estas funciones nos devolverán `TRUE` y en caso contrario devolverán `FALSE`.

```
is.numeric(5)
```

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

```
is.character(5)
```

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

5 Operadores

Los operadores son los símbolos que le indican a R que debe realizar una tarea. Combinando datos y operadores es que logramos que R haga su trabajo.

Existen operadores específicos para cada tipo de tarea. Los tipos de operadores principales son los siguientes:

- Aritméticos
- Relacionales
- Lógicos
- De asignación

Familiarizarnos con los operadores nos permitirá manipular y transformar datos de distintos tipos.

5.1 Operadores aritméticos

Como su nombre lo indica, este tipo de operador es usado para operaciones aritméticas.

En R tenemos los siguientes operadores aritméticos:

Operador	Operación	Ejemplo	Resultado
`+`	Suma	`5 + 3`	8
`-`	Resta	`5 - 3`	2
`*`	Multiplicación	`5 * 3`	18
`/`	División	`5 /3`	1.666667
`^`	Potencia	`5 ^ 3`	125
`%%`	División entera	`5 %% 3`	2

Es posible realizar operaciones aritméticas con datos de tipo **entero** y **numérico**.

Si escribes una operación aritmética en la consola de R y das Enter, esta se realiza y se devuelve su resultado.

```
15 * 3
```

```
## [1] 45
```

Cuando intentas realizar una operación aritmética con otro tipo de dato, R primero intentará coercionar ese dato a uno numérico. Si la coerción tiene éxito se realizará la operación normalmente, si falla, el resultado será un error.

Por ejemplo, `4 + "tres"` devuelve: `Error in 4 + "tres" : non-numeric argument for binary operator.`

```
4 + "tres"
```

```
## Error in 4 + "tres": non-numeric argument to binary operator
```

El mensaje *"non-numeric argument for binary operator"* aparece siempre que intentas realizar una operación aritmética con un argumento no numérico. Si te encuentras un un error que contiene este mensaje, es la primera pista para que identifiques donde ha ocurrido un problema.

Cualquier operación aritmética que intentemos con un dato `NA`, devolverá `NA` como resultado.

```
NA - 66
```

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

```
21 * NA
```

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

```
NA ^ 13
```

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

5.1.1 La división entera

Entre los operadores aritméticos, el de división entera o **módulo** requiere una explicación adicional sobre su uso. La operación que realiza es una división de un número entre otro, pero en lugar de devolver el cociente, nos devuelve el residuo.

Por ejemplo, si hacemos una división entera de 4 entre 2, el resultado será 0. Esta es una división exacta y no tiene residuo.

```
4 %% 2
```

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

En cambio, si hacemos una división entera de 5 entre 2, el resultado será 1, pues este es el residuo de la operación.

```
5 %% 2
```

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

5.2 Operadores relacionales

Los operadores lógicos son usados para hacer comparaciones y siempre devuelven como resultado `TRUE` o `FALSE` (verdadero o falso, respectivamente).

Operador	Comparación	Ejemplo	Resultado
`<`	Menor que	`5 < 3`	`FALSE`
`<=`	Menor o igual que	`5 <= 3`	`FALSE`
`>`	Mayor que	`5 > 3`	`TRUE`
`>=`	Mayor o igual que	`5 >= 3`	`TRUE`
`==`	Exactamente igual que	`5 == 3`	`FALSE`
`!=`	No es igual que	`5 != 3`	`TRUE`

Es posible comparar cualquier tipo de dato sin que resulte en un error.

Sin embargo, al usar los operadores `>`, `>=`, `<` y `<=` con cadenas de texto, estos tienen un comportamiento especial.

Por ejemplo, `"casa" > "barco"` nos devuelve `TRUE`.

```
"casa" > "barco"
```

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

Este resultado se debe a que se ha hecho una comparación por orden alfabético. En este caso, la palabra "casa" tendría una posición posterior a "barco", pues empieza con "c" y esta letra tiene una posición posterior a la "b" en el alfabeto. Por lo tanto, es verdadero que sea "mayor".

Cuando intentamos comparar factores, siempre obtendremos como resultado `NA` y una advertencia acerca de que estos operadores no son significativos para datos de tipo factor.

```
as.factor("casa") > "barco"
```

```
## Warning in Ops.factor(as.factor("casa"), "barco"): '>' not meaningful for
## factors
```

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

5.3 Operadores lógicos

Los operadores lógicos son usados para operaciones de **álgebra Booleana**, es decir, para describir relaciones lógicas, expresadas como verdadero (`TRUE`) o falso (`FALSO`).

Operador	Comparación	Ejemplo	Resultado
`x | y`	x Ó y es verdadero	`TRUE | FALSE`	`TRUE`
`x & y`	x Y y son verdaderos	`TRUE & FALSE`	`FALSE`
`!x`	x no es verdadero (negación)	`!TRUE`	`FALSE`
`isTRUE(x)`	x es verdadero (afirmación)	`isTRUE(TRUE)`	`TRUE`

Los operadores `|` y `&` siguen estas reglas:

- `|` devuelve `TRUE` si alguno de los datos es `TRUE`
- `&` solo devuelve `TRUE` si ambos datos es `TRUE`
- `|` solo devuelve `FALSE` si ambos datos son `FALSE`
- `&` devuelve `FALSE` si alguno de los datos es `FALSE`

Estos operadores pueden ser usados con estos con datos de tipo **numérico**, **lógico** y **complejo**. Al igual que con los operadores relacionales, los operadores lógicos siempre devuelven `TRUE` o `FALSE`.

Para realizar operaciones lógicas, todos los valores numéricos y complejos distintos a `0` son coercionados a `TRUE`, mientras que `0` siempre es coercionado a `FALSE`.

Por ejemplo, `5 | 0` resulta en `TRUE` y `5 & FALSE` resulta en `FALSE`. Podemos comprobar lo anterior con la función `isTRUE()`.

```
5 | 0
```

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

```
5 & 0
```

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

```
isTRUE(0)
```

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

```
isTRUE(5)
```

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

Estos operadores se pueden combinar para expresar relaciones complejas.

Por ejemplo, la negación `FALSE` Y `FALSE` dará como resultado `TRUE`.

```
!(FALSE | FALSE)
```

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

También podemos combinar operadores lógicos y relacionales, dado que esto últimos dan como resultado `TRUE` y `FALSE`.

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

5.4 Operadores de asignación

Este es probablemente el operador más importante de todos, pues nos permite asignar datos a variables.

Operador	Operación
`<-`	Asigna un valor a una variable
`=`	Asigna un valor a una variable

Aunque podemos usar el signo igual para una asignación, a lo largo de este libro utilizaremos `<-`, por ser característico de R y fácil de reconocer visualmente.

Después de realizar la operación de asignación, podemos usar el nombre de la variable para realizar operaciones con ella, como si fuera del tipo de datos que le hemos asignado. Si asignamos un valor a una variable a la que ya habíamos asignado datos, nuestra variable conserva el valor más reciente.

Además, esta operación nos permite "guardar" el resultado de operaciones, de modo que podemos recuperarlos sin necesidad de realizar las operaciones otra vez. Basta con llamar el nombre de la variable en la consola

En este ejemplo, asignamos valores a las variables `estatura` y `peso`.

```
estatura <- 1.73
peso <- 83
```

Llamamos a sus valores asignados

```
estatura
```

```
## [1] 1.73
```

```
peso
```

```
## [1] 83
```

Usamos los valores asignados para realizar operaciones.

```
peso / estatura ^ 2
```

```
## [1] 27.7323
```

Cambiamos el valor de una variable a uno nuevo y realizamos operaciones

```
peso <- 76
```

```
peso
```

```
## [1] 76
```

```
peso / estatura ^ 2
```

```
## [1] 25.39343
```

```
estatura <- 1.56
peso <- 48
```

```
peso / estatura ^ 2
```

```
## [1] 19.72387
```

Asignamos el resultado de una operación a una variable nueva.

```
bmi <- peso / estatura ^ 2
```

```
bmi
```

```
## [1] 19.72387
```

Como podrás ver, es posible asignar a una variable valores de otra variable o el resultado de operaciones con otras variables.

```
velocidad_inicial <- 110
velocidad_final <- 185

tiempo_inicial <- 0
tiempo_final <- 15

variacion_velocidad <- velocidad_final - velocidad_inicial
variacion_tiempo <- tiempo_final - tiempo_inicial

variacion_velocidad / variacion_tiempo
```

```
## [1] 5
```

5.5 Orden de operaciones

En R, al igual que en matemáticas, las operaciones tienen un orden de evaluación definido.

Cuanto tenemos varias operaciones ocurriendo al mismo tiempo, en realidad, algunas de ellas son realizadas antes que otras y el resultado de ellas dependerá de este orden.

El orden de operaciones incluye a las aritméticas, relacionales, lógicas y de asignación.

En la tabla siguiente se presenta el orden en que ocurren las operaciones que hemos revisado en este capítulo.

Orden	Operadores
1	^
2	* /
3	+ -
4	< > <= >= == !=
5	!
6	&
7	|
8	<-

Si deseamos que una operación ocurra antes que otra, rompiendo este orden de evaluación, usamos paréntesis.

Podemos tener paréntesis anidados.

6 Estructuras de datos

Las estructuras de datos son objetos que contienen datos. Cuando trabajamos con R, lo que estamos haciendo es manipular estas estructuras.

Las estructuras tienen diferentes características. Entre ellas, las que distinguen a una estructura de otra son su número de **dimensiones** y si son **homogeneas** o **hereterogeneas**.

La siguiente tabla muestra las principales estructuras de control que te encontrarás en R.

Dimensiones	Homogeneas	Heterogeneas
1	Vector	Lista
2	Matriz	Data frame
n	Array	

Adaptado de Wickham (2016).

Veamos las características de cada una de ellas.

6.1 Vectores

Un vector es la estructura de datos más sencilla en R. Un vector es una colección de uno o más datos del mismo tipo.

Todos los vectores tienen tres propiedades:

- **Tipo**. Un vector tiene el mismo tipo que los datos que contiene. Si tenemos un vector que contiene datos de tipo numérico, el vector será también de tipo numérico. Los vectores son **atómicos**, pues sólo pueden contener datos de un sólo tipo, no es posible mezclar datos de tipos diferentes dentro de ellos.
- **Largo**. Es el número de elementos que contiene un vector. El largo es la única **dimensión** que tiene esta estructura de datos.
- **Atributos**. Los vectores pueden tener metadatos de muchos tipos, los cuales describen características de los datos que contienen. Todos ellos son incluidos en esta propiedad. En este libro no se usarán vectores con metadatos, por ser una propiedad con usos van más allá del alcance de este libro.

Cuando una estructura únicamente puede contener datos de un sólo tipo, como es el caso de los vectores, decimos que es **homogénea**, pero no implica que necesariamente sea **atómica**. Regresaremos sobre esto al hablar de matrices y arrays.

Como los vectores son la estructura de datos más sencilla de R, datos simples como el número **3**, son en realidad vectores. En este caso, un vector de tipo numérico y largo igual a 1.

```
3
```

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

Verificamos que el **3** es un vector con la función `is.vector()`.

```
is.vector(3)
```

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

Y usamos la función `length()` para conocer su largo.

```r
length(3)
```

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

Lo mismo ocurre con los demás tipos de datos, por ejemplo, con cadenas de texto y datos lógicos.

```r
is.vector("tres")
```

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

```r
is.vector(TRUE)
```

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

6.1.1 Creación de vectores

Creamos vectores usando la función `c()` (*combinar*).

Llamamos esta función y le damos como argumento los elementos que deseamos combinar en un vector, separados por comas.

```r
# Vector numérico
c(1, 2, 3, 5, 8, 13)
```

```
## [1]  1  2  3  5  8 13
```

```r
# Vector de cadena de texto
c("arbol", "casa", "persona")
```

```
## [1] "arbol"   "casa"    "persona"
```

```r
# Vector lógico
c(TRUE, TRUE, FALSE, FALSE, TRUE)
```

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

Si deseamos agregar un elemento a un vector ya existente, podemos hacerlo combinando nuestro vector original con los elementos nuevos y asignando el resultado a nuestro vector original.

```r
mi_vector <- c(TRUE, FALSE, TRUE)

mi_vector <- c(mi_vector, FALSE)

mi_vector
```

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

Naturalmente, podemos crear vectores que son combinación de vectores.

```r
mi_vector_1 <- c(1, 3, 5)
mi_vector_2 <- c(2, 4, 6)

mi_vector_3 <- c(mi_vector_1, mi_vector_2)

mi_vector_3
```

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

Si intentamos combinar datos de diferentes tipos en un mismo vector, R realizará coerción automáticamente. El vector resultante será del tipo más flexible entre los datos que contenga, siguiendo las reglas de ***coerción***.

Creamos un vector numérico.

```
mi_vector <- c(1, 2, 3)
class(mi_vector)
```

```
## [1] "numeric"
```

Si intentamos agregar un dato de tipo cadena de texto, nuestro vector ahora será de tipo cadena de texto.

```
mi_vector_nuevo <- c(mi_vector, "a")
class(mi_vector_nuevo)
```

```
## [1] "character"
```

Como las cadenas de texto son el tipo de dato más flexible, siempre que creamos un vector que incluye un dato de este tipo, el resultado será un vector de texto.

```
mi_vector_mezcla <- c(FALSE, 2, "tercero", 4.00)
class(mi_vector_mezcla)
```

```
## [1] "character"
```

Podemos crear vectores de secuencias numéricas usando :. De un lado de los dos puntos escribimos el número de inicio de la secuencia y del otro el final.

Por ejemplo, creamos una secuencia del 1 al 10.

```
1:10
```

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

También podemos crear una secuencia del 10 al 1.

```
10:1
```

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

Las secuencias creadas con : son consecutivas con incrementos o decrementos de 1. Estas secuencias pueden empezar con cualquier número, incluso si este es negativo o tiene cifras decimales

```
# Número negativo
-43:-30
```

```
##  [1] -43 -42 -41 -40 -39 -38 -37 -36 -35 -34 -33 -32 -31 -30
```

```
# Número con cifras decimales
67.23:75
```

```
## [1] 67.23 68.23 69.23 70.23 71.23 72.23 73.23 74.23
```

Si nuestro número de inicio tiene cifras decimales, estas serán respetadas al hacer los incrementos o decrementos de uno en uno. En contraste, si es nuestro número de final el que tiene cifras decimales, este será redondeado.

```
# Se conservan los decimales del inicio
-2.48:2
```

```
## [1] -2.48 -1.48 -0.48  0.52  1.52
```

```
56.007:50
```

```
## [1] 56.007 55.007 54.007 53.007 52.007 51.007 50.007
```

```
# Se redondean los decimales del final
166:170.05
```

```
## [1] 166 167 168 169 170
```

```
968:960.928
```

```
## [1] 968 967 966 965 964 963 962 961
```

6.1.2 Vectorización de operaciones

Existen algunas operaciones al aplicarlas a un vector, se aplican a cada uno de sus elementos. A este proceso le llamamos **vectorización**.

Por ejemplo, las operaciones aritméticas pueden vectorizarse. Si las aplicamos a un vector, la operación se realizará para cada uno de los elementos que contiene.

```
# Nuestro vector
mi_vector <- c(2, 3, 6, 7, 8, 10, 11)

# Las operaciones
mi_vector + 2
```

```
## [1]  4  5  8  9 10 12 13
```

```
mi_vector * 2
```

```
## [1]  4  6 12 14 16 20 22
```

```
mi_vector / 2
```

```
## [1] 1.0 1.5 3.0 3.5 4.0 5.0 5.5
```

```
mi_vector ^ 2
```

```
## [1]   4   9  36  49  64 100 121
```

```
mi_vector %% 2
```

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

Esta manera de aplicar una operación es muy eficiente. Comparada con otros procedimientos, requiere de menos tiempo de cómputo, lo cual a veces es considerable, en particular cuando trabajamos con un número grande de datos.

Aunque el nombre de este proceso es **vectorización**, también funciona, en ciertas circunstancias, para otras estructuras de datos.

6.2 Matrices y arrays

Las matrices y arrays pueden ser descritas como **vectores multidimensionales**. Al igual que un vector, únicamente pueden contener datos de un sólo tipo, pero además de largo, tienen más dimensiones.

En un sentido estricto, las matrices son una caso especial de un array, que se distingue por tener **específicamente dos dimensiones**, un "largo"" y un "alto". Las matrices son, por lo tanto, una estructura con forma rectangular, con renglones y columnas.

Como las matrices son usadas de manera regular en matemáticas y estadística, es una estructura de datos de uso común en R común y en la que nos enfocaremos en este libro.

Los arrays, por su parte, pueden tener un número arbitrario de dimensiones. Pueden ser cubos, hipercubos y otras formas. Su uso no es muy común en R, aunque a veces es deseable contar con objetos n-dimensionales para manipular datos. Como los arrays tienen la restricción de que todos sus datos deben ser del mismo tipo, no importando en cuántas dimensiones se encuentren, esto limita sus usos prácticos.

En general, es preferible usar listas en lugar de arrays, una estructura de datos que además tienen ciertas ventajas que veremos más adelante.

6.2.1 Creación de matrices

Creamos matrices en R con la función `matrix()`. La función `matrix()` acepta dos argumentos, `nrow` y `ncol`. Con ellos especificamos el número de renglones y columnas que tendrá nuestra matriz.

```
# Un vector numérico del uno al doce
1:12
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
# matrix() sin especificar renglones ni columnas
matrix(1:12)
```

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

```
# Tres renglones y cuatro columnas
matrix(1:12, nrow = 3, ncol = 4)
```

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

```
# Cuatro columnas y tres columnas
matrix(1:12, nrow = 4, ncol = 3)
```

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

```
# Dos renglones y seis columnas
matrix(1:12, nrow = 4, ncol = 3)
```

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

Los datos que intentemos agrupar en una matriz serán acomodados en orden, de arriba a abajo, y de izquierda a derecha, hasta formar un rectángulo.

Si multiplicamos el número de renglones por el número de columnas, obtendremos el número de celdas de la matriz. En los ejemplo anteriores, el número de celdas es igual al número de elementos que queremos acomodar, así que la operación ocurre sin problemas.

Cuando intentamos acomodar un número diferente de elementos y celdas, ocurren dos cosas diferentes.

Si el número de elementos es mayor al número de celdas, se acomodarán todos los datos que sean posibles y los demás se omitirán.

```
matrix(1:12, nrow = 3, ncol = 3)
```

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

Si, por el contrario, el número de celdas es mayor que el número de elementos, estos se **reciclaran**. En cuanto los elementos sean insuficientes para acomodarse en las celdas, R nos devolverá una advertencia y se empezaran a usar los elementos a partir del primero de ellos

```
matrix(1:12, nrow = 5, ncol = 4)
```

```
## Warning in matrix(1:12, nrow = 5, ncol = 4): data length [12] is not a sub-
## multiple or multiple of the number of rows [5]
```

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

Otro procedimiento para crear matrices es la unión vectores con las siguientes funciones:

- `cbind()` para unir vectores, usando cada uno como una columna.

- `rbind()` para unir vectores, usando cada uno como un renglón.

De este modo podemos crear cuatro vectores y unirlos para formar una matriz. Cada vector será un renglón en esta matriz.

Creamos cuatro vectores, cada uno de largo igual a cuatro.

```
vector_1 <- 1:4
vector_2 <- 5:8
vector_3 <- 9:12
vector_4 <- 13:16
```

Usamos `rbind()` para crear un matriz, en la que cada vector será un renglón.

```
matriz <- rbind(vector_1, vector_2, vector_3, vector_4)

# Resultado
matriz
```

```
##          [,1] [,2] [,3] [,4]
## vector_1    1    2    3    4
## vector_2    5    6    7    8
## vector_3    9   10   11   12
## vector_4   13   14   15   16
```

Si utilizamos `cbind()`, entonces cada vector será una columna.

```
matriz <- cbind(vector_1, vector_2, vector_3, vector_4)

# Resultado
matriz
```

```
##      vector_1 vector_2 vector_3 vector_4
## [1,]        1        5        9       13
## [2,]        2        6       10       14
## [3,]        3        7       11       15
## [4,]        4        8       12       16
```

Al igual que con `matrix()`, los elementos de los vectores son reciclados para formar una estructura rectangular y se nos muestra un mensaje de advertencia.

```
# Elementos de largo diferente
vector_1 <- 1:2
vector_2 <- 1:3
vector_3 <- 1:5

matriz <- cbind(vector_1, vector_2, vector_3)
```

```
## Warning in cbind(vector_1, vector_2, vector_3): number of rows of result is
## not a multiple of vector length (arg 1)
```

```
# Resultado
matriz
```

```
##      vector_1 vector_2 vector_3
## [1,]        1        1        1
## [2,]        2        2        2
## [3,]        1        3        3
## [4,]        2        1        4
## [5,]        1        2        5
```

Finalmente, las matrices pueden contener `NA`s.

Creamos dos vectores con un `NA` en ellos.

```
vector_1 <- c(NA, 1, 2)
vector_2 <- c(3,  4, NA)
```

Creamos una matriz con `rbind()`.

```
matriz <- rbind(vector_1, vector_2)

# Resultados
matriz
```

```
##          [,1] [,2] [,3]
## vector_1   NA    1    2
## vector_2    3    4   NA
```

Como `NA` representa datos perdidos, puede estar presente en compañía de todo tipo de de datos.

6.2.2 Propiedades de las matrices

No obstante que las matrices y arrays son estructuras que sólo pueden contener un tipo de datos, no son atómicas. Su clase es igual a **matriz (matrix)** o **array** segun corresponda.

Verificamos esto usando la función `class()`.

```
mi_matriz <- matrix(1:10)

class(mi_matriz)
```

```
## [1] "matrix"
```

Las matrices y arrays pueden tener más de una dimensión.

Obtenemos el número de dimensiones de una matriz o array con la función `dim()`. Esta función nos devolverá varios números, cada uno de ellos indica la cantidad de elementos que tiene una dimensión.

```
mi_matriz <- matrix(1:12, nrow = 4, ncol = 3)
dim(mi_matriz)
```

```
## [1] 4 3
```

Cabe señalar que si usamos `dim()` con un vector, obtenemos `NULL`. Esto ocurre con todos los objetos unidimensionales

```
mi_vector <- 1:12

dim(mi_vector)
```

```
## NULL
```

Finalmente, las operaciones aritméticas también son vectorizadas al aplicarlas a una matriz. La operación es aplicada a cada uno de los elementos de la matriz.

Creamos una matriz.

```r
mi_matriz <- matrix(1:9, nrow = 3, ncol = 3)

# Resultado
mi_matriz
```

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

Intentemos sumar, multiplicar y elevar a la tercera potencia.

```r
# Suma
mi_matriz + 1
```

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

```r
# Multiplicación
mi_matriz * 2
```

```
##      [,1] [,2] [,3]
## [1,]    2    8   14
## [2,]    4   10   16
## [3,]    6   12   18
```

```r
# Potenciación
mi_matriz ^ 3
```

```
##      [,1] [,2] [,3]
## [1,]    1   64  343
## [2,]    8  125  512
## [3,]   27  216  729
```

Si intentamos vectorizar una operación utilizando una matriz con `NA`s, esta se aplicará para los elementos válidos, devolviendo `NA` cuando corresponda.

Creamos una matriz con `NA`s.

```r
vector_1 <- c(NA, 2, 3)
vector_2 <- c(4, 5, NA)

matriz <- rbind(vector_1, vector_2)

# Resultado
matriz
```

```
##          [,1] [,2] [,3]
## vector_1   NA    2    3
## vector_2    4    5   NA
```

Intentamos dividir sus elementos entre dos.

```r
matriz / 2
```

```
##          [,1] [,2] [,3]
## vector_1   NA  1.0  1.5
## vector_2    2  2.5   NA
```

Finalmente, podemos usar la función `t()` para transponer una matriz, es decir, rotarla 90°.

Creamos una matriz con tres renglones y dos columnas.

```
matriz <- matrix(1:6, nrow = 3)

# Resultado
matriz
```

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

Usamos `t()` para transponer.

```
matriz_t <- t(matriz)

# Resultado
matriz_t
```

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

Obtenemos una matriz con dos renglones y dos columnas.

6.3 Data frames

Los data frames son estructuras de datos de dos dimensiones (rectangulares) que pueden contener datos de diferentes tipos, por lo tanto, son heterogéneas. Esta estructura de datos es la más usada para realizar análisis de datos y seguro te resultará familiar si has trabajado con otros paquetes estadísticos.

Podemos entender a los data frames como una versión más flexible de una matriz. Mientras que en una matriz todas las celdas deben contener datos del mismo tipo, los renglones de un data frame admiten datos de distintos tipos, pero sus columnas conservan la restricción de contener datos de un sólo tipo.

En términos generales, los renglones en un data frame representan casos, individuos u observaciones, mientras que las columnas representan atributos, rasgos o variables. Por ejemplo, así lucen los primeros cinco renglones del objeto **iris**, el famoso conjunto de datos *Iris de Ronald Fisher*, que está incluido en todas las instalaciones de R.

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
```

Los primeros cinco renglones corresponden a cinco casos, en este caso flores. Las columnas son variables con los rasgos de cada flor: largo y ancho de sépalo, largo y ancho de pétalo, y especie.

Para crear un data frame usamos la función `data.frame()`. Esta función nos pedirá un número de vectores igual al número de columnas que deseemos. Todos los vectores que proporcionemos deben tener el mismo largo.

Esto es muy importante: **Un data frame está compuesto por vectores**.

Más adelante se hará evidente porque esta característica de un data frame es sumamente importante y también, cómo podemos sacarle provecho.

Además, podemos asignar un nombre a cada vector, que se convertirá en el nombre de la columna. Como todos los nombres, es recomendable que este sea claro, no ambiguo y descriptivo.

```
mi_df <- data.frame(
  "entero" = 1:4,
  "factor" = c("a", "b", "c", "d"),
  "numero" = c(1.2, 3.4, 4.5, 5.6),
  "cadena" = as.character(c("a", "b", "c", "d"))
)

mi_df
```

```
##   entero factor numero cadena
## 1      1      a    1.2      a
## 2      2      b    3.4      b
## 3      3      c    4.5      c
## 4      4      d    5.6      d
```

```
# Podemos usar dim() en un data frame
dim(mi_df)
```

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

```
# El largo de un data frame es igual a su número de columnas
length(mi_df)
```

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

```
# names() nos permite ver los nombres de las columnas
names(mi_df)
```

```
## [1] "entero" "factor" "numero" "cadena"
```

```
# La clase de un data frame es data.frame
class(data.frame)
```

```
## [1] "function"
```

Si los vectores que usamos para construir el data frame no son del mismo largo, los datos **no se reciclaran**. Se nos devolverá un error.

```
data.frame(
  "entero" = 1:3,
  "factor" = c("a", "b", "c", "d"),
  "numero" = c(1.2, 3.4, 4.5, 5.6),
  "cadena" = as.character(c("a", "b", "c", "d"))
)
```

```
## Error in data.frame(entero = 1:3, factor = c("a", "b", "c", "d"), numero = c(1.2, : arguments
```

También podemos coercionar esta matriz a un data frame.

Creamos una matriz.

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

Usamos `as.data.frame()` para coercionar una matriz a un data frame.

```
df <- as.data.frame(matriz)
```

Verificamos el resultado

```
class(df)
```

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

```
# Resultado
df
```

```
##   V1 V2 V3 V4
## 1  1  4  7 10
## 2  2  5  8 11
## 3  3  6  9 12
```

6.3.1 Propiedades de un data frame

Al igual que con una matriz, si aplicamos una operación aritmética a un data frame, esta se vectorizará.

Los resultados que obtendremos dependerán del tipo de datos de cada columna. R nos devolverá todas las advertencias que ocurran como resultado de las operaciones realizadas, por ejemplo, aquellas que hayan requerido una coerción.

```
mi_df <- data.frame(
  "entero" = 1:4,
  "factor" = c("a", "b", "c", "d"),
  "numero" = c(1.2, 3.4, 4.5, 5.6),
  "cadena" = as.character(c("a", "b", "c", "d"))
)

mi_df * 2
```

```
## Warning in Ops.factor(left, right): '*' not meaningful for factors
```

```
## Warning in Ops.factor(left, right): '*' not meaningful for factors
```

```
##   entero factor numero cadena
## 1      2     NA    2.4     NA
## 2      4     NA    6.8     NA
## 3      6     NA    9.0     NA
## 4      8     NA   11.2     NA
```

6.4 Listas

Las listas, al igual que los vectores, son estructuras de datos unidimensionales, sólo tienen largo, pero a diferencia de los vectores cada uno de sus elementos puede ser de diferente tipo o incluso de diferente clase, por lo que son estructuras heterogéneas.

Podemos tener listas que contengan datos atómicos, vectores, matrices, arrays, data frames u otras listas. Esta última característica es la razón por la que una lista puede ser considerada un vector recursivo, pues es un objeto que puede contener objetos de su misma clase.

Para crear una lista usamos la función `list()`, que nos pedirá los elementos que deseamos incluir en nuestra lista. Para esta estructura, no importan las dimensiones o largo de los elementos que queramos incluir en ella.

Al igual que con un data frame, tenemos la opción de poner nombre a cada elemento de una lista.

Por último, no es posible vectorizar operaciones aritméticas usando una lista, se nos devuelve un error como resultado.

```
mi_vector <- 1:10
mi_matriz <- matrix(1:4, nrow = 2)
mi_df     <- data.frame("num" = 1:3, "let" = c("a", "b", "c"))

mi_lista <- list("un_vector" = mi_vector, "una_matriz" = mi_matriz, "un_df" = mi_df)

mi_lista
```

```
## $un_vector
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $una_matriz
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $un_df
##   num let
## 1   1   a
## 2   2   b
## 3   3   c
```

Creamos una lista que contiene otras listas.

```
lista_recursiva <- list("lista1" = mi_lista, "lista2" = mi_lista)

# Resultado
lista_recursiva
```

```
## $lista1
## $lista1$un_vector
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $lista1$una_matriz
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $lista1$un_df
##   num let
## 1   1   a
## 2   2   b
## 3   3   c
##
##
## $lista2
## $lista2$un_vector
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $lista2$una_matriz
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $lista2$un_df
##   num let
## 1   1   a
## 2   2   b
## 3   3   c
```

6.4.1 Propiedades de una lista

Una lista es unidimensional, sólo tiene largo.

El largo de una lista es igual al número de elementos que contiene, sin importar de qué tipo o clase sean. Usamos la lista recursiva que creamos en la sección anterior para ilustrar esto.

```
length(lista_recursiva)
```

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

Dado que una lista siempre tiene una sola dimensión, la función `dim()` nos devuelve `NULL`.

```
dim(lista_recursiva)
```

```
## NULL
```

Las listas tienen clase **list**, sin importar qué elementos contienen.

```
class(lista_recursiva)
```

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

Finalmente, no es posible vectorizar operaciones aritméticas usando listas. Al intentarlo nos es devuelto un error.

```
mi_lista / 2
```

```
## Error in mi_lista/2: non-numeric argument to binary operator
```

Si deseamos aplicar una función a cada elemento de una lista, usamos `lapply()`, como veremos en el capítulo 10.

7 Subconjuntos

En R, podemos obtener subconjuntos de todas las estructuras de datos. Esto es, podemos extraer datos de las estructuras que tengan una o más características en especial.

Para esta extracción usaremos índices, operadores lógicos y álgebra Booleana. Aunque cada estructura de datos de R es diferente, existen procedimientos para obtener subconjuntos que pueden usarse con todas ellas. Por supuesto, hay otras que funcionan sólo con algunas estructuras.

7.1 Índices

Crer subconjuntos usando índices es el procedimiento más universal en R, pues funciona con todas las estructuras de datos.

Los índices en R son **posicionales**. Cuando usamos este método le pedimos a R que extraiga de una estructura los datos que se encuentran en una o varias posiciones específicas.

Escribimos corchetes `[]` después de un objeto para obtener subconjuntos con índices. Dentro de los corchetes escribimos el o los números que corresponden a la posición que nos interesa extraer del objeto. Por ejemplo:

- `objeto[3]`
- `lista[4:5]`
- `dataframe[c(2, 7), ]`

Veamos un ejemplo con un vector. Creamos un vector que contiene los nombres de distintos niveles educativos.

```
nivel <- c("Preescolar", "Primaria", "Secundaria", "Educación Media Superior",
           "Educación Superior")
```

```
nivel
```

```
## [1] "Preescolar"               "Primaria"
## [3] "Secundaria"               "Educación Media Superior"
## [5] "Educación Superior"
```

Este es un vector de largo igual a 5.

```
length(nivel)
```

```
## [1] 5
```

¿Cómo obtendríamos el tercer elemento de este vector usando índices? ¿Y del primer al cuarto elemento? ¿O el segundo y quinto elemento?

Sabemos que los índices son posicionales y que se usan corchetes para realizar la operación de extracción, por lo tanto, las respuestas a las preguntas anteriores son relativamente intuitivas.

Para extraer el tercer elemento hacemos lo siguiente.

```
nivel[3]
```

```
## [1] "Secundaria"
```

A diferencia de la mayoría de los lenguajes de programación, los índices en R empiezan en 1, no en 0. El índice del primer elemento de una estructura de datos siempre es 1.

Por lo tanto, para extraer del primer al cuarto elemento de un vector, usamos los números del 1 al 4.

```
nivel[1:4]
```

```
## [1] "Preescolar"              "Primaria"
## [3] "Secundaria"              "Educación Media Superior"
```

Cuando deseamos extraer elementos en posiciones no consecutivas, usamos vectores. Por ejemplo, para extraer el segundo y quinto elemento del vector ***nivel**, lo siguiente no funciona.

```
nivel[2, 5]
```

```
## Error in nivel[2, 5]: incorrect number of dimensions
```

Si usamos un vector, tendremos éxito en extraer el segundo y quinto elemento de **nivel**.

```
nivel[c(2, 5)]
```

```
## [1] "Primaria"           "Educación Superior"
```

¿Porqué no funcionó usar **nivel[2, 5]** para extraer dos elementos no consecutivos en nuestro vector?

El mensaje de error nos da una pista muy importante. Al usar una coma dentro de los corchetes estamos pidiendo a R que busque los índices solicitados en más de una dimensión.

Entonces, al llamar `nivel[2, 5]` le pedimos a R que extraiga el elemento que se encuentA en la posición 2 de la primera dimensión del vector, y el elemento en la posición 5 de su segunda dimensión. Como los vectores son unidimensionales, es imposible cumplir esta instrucción y se produce un error.

En cambio, usar `nivel[c(2, 5)]` funciona porque estamos dando un vector con dos números, pero ambos en una misma dimensión de nuestro objeto.

Para estructuras con más de una dimensión, los índices hacen referencia a posiciones para cada una de ellas. En estructuras de dos dimensiones (matrices y data frames), **el primer índice es para los renglones y la segunda para las columnas**.

Este es un tipo de operación muy común al trabajar con data frames y matrices

7.2 Índices, data frames y matrices

Para ilustrar cómo usar índices con objetos rectangulares, vamos a crear un data frame llamado **mi_df**.

```r
mi_df <- data.frame("nombre" = c("Armando", "Elsa", "Ignacio", "Olga"),
                    "edad" = c(20, 24, 22, 30),
                    "sexo" = c("H", "M", "M", "H"),
                    "grupo" = c(0, 1, 1, 0))

mi_df
```

```
##    nombre edad sexo grupo
## 1 Armando   20    H     0
## 2    Elsa   24    M     1
## 3 Ignacio   22    M     1
## 4    Olga   30    H     0
```

Confirmamos que nuestro data frame tiene dos dimensiones: tres renglones y tres columnas.

```r
dim(mi_df)
```

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

Con índices podemos extraer un dato que se encuentra en un renglón y columna específico.

```r
# Extraer dato en el tercer renglón y tercera columna
mi_df[3, 3]
```

```
## [1] M
## Levels: H M
```

```r
# Extraer dato en el primer renglón y segunda columna
mi_df[1, 2]
```

```
## [1] 20
```

```r
# Extraer dato en el segundo renglón y el primer renglón
mi_df[2, 1]
```

```
## [1] Elsa
## Levels: Armando Elsa Ignacio Olga
```

```r
# Extraer dato en
mi_df[1:2, 3:4]
```

```
##   sexo grupo
## 1    H     0
## 2    M     1
```

```r
# Si dejamos vacio el índice para una dimensión, nos son devueltos todos
# los datos que contiene
mi_df[ , 1]
```

```
## [1] Armando Elsa    Ignacio Olga
## Levels: Armando Elsa Ignacio Olga
```

```r
mi_df[1,  ]
```

```
##    nombre edad sexo grupo
## 1 Armando   20    H     0
```

```
# Podemos usar vectores para elegir elementos no consecutivos
mi_df[c(1, 3),  c(1, 4)]
```

```
##    nombre grupo
## 1 Armando     0
## 3 Ignacio     1
```

Para objetos de tres o más dimensiones se siguen las mismas reglas, aunque ya no es tan fácil hablar de renglones y columnas. Por ejemplo, un array de cuatro dimensiones.

```
mi_array <- array(data = 1:16, dim = c(2, 2, 2, 2))

# Comprobamos las dimensiones
dim(mi_array)
```

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

```
# Así luce un array sencillo de cuatro dimensiones.
mi_array
```

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

```
# Un par de subconjuntos
mi_array[1, 2, 1, 2]
```

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

```
mi_array[ , 1, 1, 1]
```

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

Finalmente, conviene saber que la posición de los elementos en una estructura se determinan en su creación.

```
vector1 <- c("a", "b", "c")
vector2 <- c("a", "c", "b")

# Aunque los vectores tienen los mismos elementos, su orden es diferente
vector1[2]
```

```
## [1] "b"
```

```
vector2[2]
```

```
## [1] "c"
```

Esto es suficiente por ahora. La siguiente ocasión continuaremos con más métodos para crear subconjuntos.

Esta ocasión terminaremos de revisar las formas de extraer subconjuntos de nuestros datos. Puede parecer que le estamos dedicando mucho tiempo a este tema, pero es crucial para ahorrarnos dolores de cabeza en el futuro.

7.3 Subconjuntos por nombre

Podemos usar los nombres de los elementos en una estructura de datos para extraer subconjuntos. Este método es usado principalmente para trabajar con data frames y listas.

Al igual que con los índices, podemos usar corchetes cuadrados (**[ ]**) para obtener subconjuntos, pero en lugar de escribir un número de índice, escribimos el nombre del elemento que deseamos extraer como una cadena de texto, es decir, entre comillas.

Por ejemplo, usando el mismo data frame que la ocasión anterior.

```
mi_df <- data.frame("nombre" = c("Armando", "Elsa", "Ignacio", "Olga"),
                    "edad" = c(20, 24, 22, 30), "sexo" = c("H", "M", "M", "H"),
                    "grupo" = c(0, 1, 1, 0))

mi_df["nombre"]
```

```
##    nombre
## 1 Armando
## 2    Elsa
## 3 Ignacio
## 4    Olga
```

```
mi_df["grupo"]
```

```
##   grupo
## 1     0
## 2     1
## 3     1
## 4     0
```

En una data frame, cada uno de sus elementos representa una columna en los datos, así que en realidad estamos pidiendo a R que nos devuelva las columnas con los nombres que le indicamos.

Podemos extraer más de un elemento si en lugar de una cadena de texto escribimos un vector de texto entre los corchetes.

```
# Esto funciona
mi_df[c("edad", "sexo")]
```

```
##   edad sexo
## 1   20    H
## 2   24    M
## 3   22    M
## 4   30    H
```

```
# Esto no funciona como esperamos
mi_df["edad", "sexo"]
```

```
## [1] <NA>
## Levels: H M
```

```
# Además, las columnas son devueltas en el orden que las pedimos
mi_df[c("sexo", "edad")]
```

```
##   sexo edad
## 1    H   20
## 2    M   24
## 3    M   22
## 4    H   30
```

Si pedimos un nombre que no existe en nuestros datos, obtenemos un error.

```
mi_df["calificacion"]
```

```
## Error in `[.data.frame`(mi_df, "calificacion"): undefined columns selected
```

Para una lista, el procedimiento es el mismo que con un data frame. En lugar de obtener columnas, obtenemos los elementos contenidos en la lista para los que hemos proporcionado un nombre.

Una diferencia importante con los data frame es que si pedimos un nombre que no existe en la lista, se nos devuelve **NULL** en lugar de un error.

```
mi_lista <- list("uno" = 1, "dos" = "2", "tres" = as.factor(3),
                 "cuatro" = c(1:4))

mi_lista["dos"]
```

```
## $dos
## [1] "2"
```

```
mi_lista[c("cuatro", "tres")]
```

```
## $cuatro
## [1] 1 2 3 4
##
## $tres
## [1] 3
## Levels: 3
```

```
mi_lista["cinco"]
```

```
## $<NA>
## NULL
```

También podemos usar el signo de dólar `$` para extraer subconjuntos usando nombres.

Este método permite extraer sólo un elemento a la vez y en un data frame, siempre devolverá una columna. Si lo deseamos, podemos escribir el nombre que nos interesa obtener sin comillas.

```
# Esto funciona
mi_df$nombre
```

```
## [1] Armando Elsa    Ignacio Olga
## Levels: Armando Elsa Ignacio Olga
```

```
# Esto también funciona
mi_df$"nombre"
```

```
## [1] Armando Elsa    Ignacio Olga
## Levels: Armando Elsa Ignacio Olga
```

```
# Esto no funciona
mi_df$c("nombre", "edad")
```

```
## Error in eval(expr, envir, enclos): attempt to apply non-function
```

Notarás que la salida al usar este método es diferente que si usamos corchetes. Si revisas la **Tarea 01**, verás más ejemplos de este comportamiento.

En pocas palabras, **distintos métodos para obtener subconjuntos pueden devolver resultados diferentes**. Más adelante en este documento veremos porqué ocurre esto

** Nombres para extraer renglones De igual manera que con los índices, si escribimos dentro de un corchete nombres separados por comas, R interpretará esto como que estamos buscando elementos en más de una dimensión.

En un data frame, el primer nombre corresponderá a renglones y el segundo a columnas. Esto funciona porque en un data frame los renglones también pueden tener nombre.

En caso de que pidamos un nombre de renglón que no existe, nos es devuelto **NA**

Por ejemplo, en el conjunto de datos **iris** los nombres de los renglones son igual a su número de renglón.

```
# Pedimos el renglón llamado "4"
iris["4", ]
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 4          4.6         3.1          1.5         0.2  setosa
```

```
# En este caso, lo anterior es equivalente a pedir el índice cuatro
iris[4, ]
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 4          4.6         3.1          1.5         0.2  setosa
```

Lo anterior nos permite extraer subconjuntos muy específicos de nuestros datos.

```
iris["110", "Species"]
```

```
## [1] virginica
## Levels: setosa versicolor virginica
```

```
iris["15", c("Sepal.Length", "Sepal.Width")]
```

```
##    Sepal.Length Sepal.Width
## 15          5.8           4
```

```
iris[c("88", "96"), c("Sepal.Length","Sepal.Width")]
```

```
##    Sepal.Length Sepal.Width
## 88          6.3         2.3
## 96          5.7         3.0
```

7.4 Subconjuntos por índice y nombre

Al extraer subconjuntos combinar índices con nombres dentro del mismo corchete.

Esta es una herramienta muy poderosa al manipular datos pues nos proporciona una flexibilidad para extrar subconjuntos.

Por supuesto, tenemos que seguir las reglas de ambos métodos.

```
iris[5:6, "Species"]
```

```
## [1] setosa setosa
## Levels: setosa versicolor virginica
```

```
iris["76", 2:3]
```

```
##    Sepal.Width Petal.Length
## 76           3          4.4
```

```
iris[c(1:2, 149:150), c("Petal.Width", "Species")]
```

```
##     Petal.Width   Species
## 1           0.2    setosa
## 2           0.2    setosa
```

```
## 149         2.3 virginica
## 150         1.8 virginica
```

Esto no funciona con las listas, Por ser estructuras de datos unidimensionales.

** Subconjuntos con su clase y tipo original: usando el signo de dólar $ y el corchete doble [[ ]]

¿Porqué llamar **mi_df["nombre"]** devuelve un resultado diferente de llamar **mi_df$nombre**?

```
mi_df["nombre"]
```

```
##    nombre
## 1 Armando
## 2    Elsa
## 3 Ignacio
## 4    Olga
```

```
mi_df$nombre
```

```
## [1] Armando Elsa    Ignacio Olga
## Levels: Armando Elsa Ignacio Olga
```

Para entender que está ocurriendo, recordemos que **un data frame está formado por vectores**. Estos vectores nunca dejan de ser vectores, aunque estén contenidos dentro de un data frame, por lo tanto, es posible extraerlos de esta estructura de datos.

Además, necesitamos entender que **cuando extraemos un subconjunto de un objeto usando corchetes, obtenemos como resultado un objeto de su misma clase**.

Esto suena a un trabalenguas, pero todo lo que quiere decir es que extraemos un subconjunto de una lista, obtenemos una lista; si lo hacemos de un data frame, obtenemos un data frame; y lo mismo para todas las estructuras de datos.

Esto lo podemos comprobar fácilmente usando la función **class( )**

```
class(mi_df)
```

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

```
class(mi_df["nombre"])
```

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

Esto cambia cuando usamos el signo de dólar **$** para extraer un subconjunto. **Si usamos un signo de dolar, obtenemos un objeto de la misma clase y tipo que era ese elemento originalmente**.

En el caso de un data frame, usar el signo de dólar siempre resulta en vectores atómicos.

```
class(mi_df$nombre)
```

```
## [1] "factor"
```

```r
class(mi_df$edad)
```

```
## [1] "numeric"
```

```r
class(mi_df$sexo)
```

```
## [1] "factor"
```

Con las listas, podemos obtener objetos de cualquier clase y tipo.

```r
# Una lista con una matriz y un data frame.
mi_lista <- list("uno" = matrix(2:2, nrow = 2),
                 "dos" = data.frame("a" = 1:2, "b" = 3:4))

# Devuelve una lista
class(mi_lista["uno"])
```

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

```r
# Devuelve una matriz
class(mi_lista$uno)
```

```
## [1] "matrix"
```

```r
# Devuelve un data frame
class(mi_lista$dos)
```

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

Otra manera de extraer elementos de un objeto con su clase original es usar corchetes dobles **[[ ]]**.

La ventaja de usar este método es que podemos usar índices y nombres dentro de los corchetes dobles, lo que nos da acceso a una mayor flexibilidad para extraer subconjuntos, además de que nos permite usarlos en estructuras de datos que tienen elementos sin nombre.

```r
mi_df[["edad"]]
```

```
## [1] 20 24 22 30
```

```r
mi_df[[2]]
```

```
## [1] 20 24 22 30
```

```r
mi_matriz <- matrix(1:9, nrow = 3, ncol = 3)

mi_matriz
```

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

```
mi_matriz[[1, 2]]
```

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

Sin embargo, no podemos usar vectores dentro los corchetes dobles para extraer subconjuntos, pues este método busca un elemento a la vez.

```
mi_matriz[[c(1, 3), 2]]
```

```
## Error in mi_matriz[[c(1, 3), 2]]: attempt to select more than one element in get1index
```

Obtener un objeto del tipo correcto al extraer un subconjunto es sumamente importante, en particular si deseamos usar este subconjunto para realizar otras operaciones.

Por ejemplo, si queremos obtener la media de la columna **Sepal.Width** del conjunto de datos iris, usamos la función **mean( )**. Sin embargo, esta función nos pide que demos como argumento un vector, de modo que debemos estar atentos a cómo extraemos la columna que nos interesa.

```
# Esto no funciona, porque estamos extrayendo un data frame
mean(iris["Sepal.Width"])
```

```
## Warning in mean.default(iris["Sepal.Width"]): argument is not numeric or
## logical: returning NA
```

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

```
# Esto sí funciona
mean(iris$Sepal.Width)
```

```
## [1] 3.057333
```

```
# Esto también funciona
mean(iris[["Sepal.Width"]])
```

```
## [1] 3.057333
```

7.5 Subconjuntos de un data frame usando condicionales

Supongamos que nos interesa obtener un subconjunto de datos que cumplen con una o más condiciones específicas.

Por ejemplo, queremos obtener todos los datos de una encuesta que corresponden a mujeres, o a personas que viven en una entidad específica o que tienen un ingreso superior a la media.

Si tenemos columnas que contengan esa información en nuestro conjunto de datos, podemos extraer subconjuntos usando condicionales dentro de los corchetes.

Esta operación tiene la siguiente estructura.

** objeto[condicion, columnas_devueltas] **

Veamos un ejemplo.

Extraeremos del conjunto iris...

... todos los casos en los que el ancho del pétalo es mayor a 2.

```
iris[iris["Petal.Width"] > 2, ]
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 101          6.3         3.3          6.0         2.5 virginica
## 103          7.1         3.0          5.9         2.1 virginica
## 105          6.5         3.0          5.8         2.2 virginica
## 106          7.6         3.0          6.6         2.1 virginica
## 110          7.2         3.6          6.1         2.5 virginica
## 113          6.8         3.0          5.5         2.1 virginica
## 115          5.8         2.8          5.1         2.4 virginica
## 116          6.4         3.2          5.3         2.3 virginica
## 118          7.7         3.8          6.7         2.2 virginica
## 119          7.7         2.6          6.9         2.3 virginica
## 121          6.9         3.2          5.7         2.3 virginica
## 125          6.7         3.3          5.7         2.1 virginica
## 129          6.4         2.8          5.6         2.1 virginica
## 133          6.4         2.8          5.6         2.2 virginica
## 136          7.7         3.0          6.1         2.3 virginica
## 137          6.3         3.4          5.6         2.4 virginica
## 140          6.9         3.1          5.4         2.1 virginica
## 141          6.7         3.1          5.6         2.4 virginica
## 142          6.9         3.1          5.1         2.3 virginica
## 144          6.8         3.2          5.9         2.3 virginica
## 145          6.7         3.3          5.7         2.5 virginica
## 146          6.7         3.0          5.2         2.3 virginica
## 149          6.2         3.4          5.4         2.3 virginica
```

... todos los casos en los que la especie sea "setosa"

```
iris[iris[["Species"]] == "setosa", ]
```

```
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1           5.1         3.5          1.4         0.2  setosa
## 2           4.9         3.0          1.4         0.2  setosa
## 3           4.7         3.2          1.3         0.2  setosa
## 4           4.6         3.1          1.5         0.2  setosa
## 5           5.0         3.6          1.4         0.2  setosa
## 6           5.4         3.9          1.7         0.4  setosa
## 7           4.6         3.4          1.4         0.3  setosa
## 8           5.0         3.4          1.5         0.2  setosa
## 9           4.4         2.9          1.4         0.2  setosa
## 10          4.9         3.1          1.5         0.1  setosa
## 11          5.4         3.7          1.5         0.2  setosa
## 12          4.8         3.4          1.6         0.2  setosa
## 13          4.8         3.0          1.4         0.1  setosa
## 14          4.3         3.0          1.1         0.1  setosa
## 15          5.8         4.0          1.2         0.2  setosa
## 16          5.7         4.4          1.5         0.4  setosa
## 17          5.4         3.9          1.3         0.4  setosa
## 18          5.1         3.5          1.4         0.3  setosa
## 19          5.7         3.8          1.7         0.3  setosa
## 20          5.1         3.8          1.5         0.3  setosa
## 21          5.4         3.4          1.7         0.2  setosa
## 22          5.1         3.7          1.5         0.4  setosa
## 23          4.6         3.6          1.0         0.2  setosa
## 24          5.1         3.3          1.7         0.5  setosa
## 25          4.8         3.4          1.9         0.2  setosa
```

```
## 26          5.0         3.0          1.6         0.2  setosa
## 27          5.0         3.4          1.6         0.4  setosa
## 28          5.2         3.5          1.5         0.2  setosa
## 29          5.2         3.4          1.4         0.2  setosa
## 30          4.7         3.2          1.6         0.2  setosa
## 31          4.8         3.1          1.6         0.2  setosa
## 32          5.4         3.4          1.5         0.4  setosa
## 33          5.2         4.1          1.5         0.1  setosa
## 34          5.5         4.2          1.4         0.2  setosa
## 35          4.9         3.1          1.5         0.2  setosa
## 36          5.0         3.2          1.2         0.2  setosa
## 37          5.5         3.5          1.3         0.2  setosa
## 38          4.9         3.6          1.4         0.1  setosa
## 39          4.4         3.0          1.3         0.2  setosa
## 40          5.1         3.4          1.5         0.2  setosa
## 41          5.0         3.5          1.3         0.3  setosa
## 42          4.5         2.3          1.3         0.3  setosa
## 43          4.4         3.2          1.3         0.2  setosa
## 44          5.0         3.5          1.6         0.6  setosa
## 45          5.1         3.8          1.9         0.4  setosa
## 46          4.8         3.0          1.4         0.3  setosa
## 47          5.1         3.8          1.6         0.2  setosa
## 48          4.6         3.2          1.4         0.2  setosa
## 49          5.3         3.7          1.5         0.2  setosa
## 50          5.0         3.3          1.4         0.2  setosa
```

... la especie de los casos en que los que el ancho del sépalo sea menor a 3.

```
iris[iris$Sepal.Width < 3, "Species"]
```

```
##  [1] setosa     setosa     versicolor versicolor versicolor versicolor
##  [7] versicolor versicolor versicolor versicolor versicolor versicolor
## [13] versicolor versicolor versicolor versicolor versicolor versicolor
## [19] versicolor versicolor versicolor versicolor versicolor versicolor
## [25] versicolor versicolor versicolor versicolor versicolor versicolor
## [31] versicolor versicolor versicolor versicolor versicolor versicolor
## [37] virginica  virginica  virginica  virginica  virginica  virginica
## [43] virginica  virginica  virginica  virginica  virginica  virginica
## [49] virginica  virginica  virginica  virginica  virginica  virginica
## [55] virginica  virginica  virginica
## Levels: setosa versicolor virginica
```

... el ancho y largo del pétalo de los casos en los que el largo del sépalo es mayor o igual a 7.2.

```
iris[iris["Sepal.Length"] >= 7.2, c("Petal.Length", "Petal.Width")]
```

```
##     Petal.Length Petal.Width
## 106          6.6         2.1
## 108          6.3         1.8
## 110          6.1         2.5
## 118          6.7         2.2
## 119          6.9         2.3
## 123          6.7         2.0
## 126          6.0         1.8
## 130          5.8         1.6
## 131          6.1         1.9
## 132          6.4         2.0
## 136          6.1         2.3
```

¿Qué es lo que está ocurriendo?

Dentro del corchete escribimos antes de una coma un subconjunto, al cual aplicamos una operación relacional. Todos los renglones en los que el resultado de esta operación sea **TRUE**, formarán parte de nuestro subconjunto.

Si no indicamos qué columnas queremos que se nos devuelvan, obtendremos todas. De esta manera podemos extraer subconjuntos que cumplen una condición, pero sólo para una columna específica.

Para entender porqué escribimos una operación relacional aplicada a un subconjunto dentro de los corchetes, nos conviene saber que las operaciones relacionales también se vectorizan.

Si al vector **iris$Petal.Width** aplicamos la operación **> 6**, esta se aplicará a todos sus elementos, devolviendo **TRUE** o **FALSE**, según corresponda.

```
as.vector(iris["Petal.Width"] > 6)
```

```
##   [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [45] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [56] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [78] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [89] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [100] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [111] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [122] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [144] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Así, lo que pedimos antes de la coma dentro de un corchete, son todos los renglones para los que la condición es verdadera.

Si quisieras, podrías elegir manualmente renglones de un data frame usando TRUE y FALSE.

```
# EL vector de datos lógicos se reciclará, así que obtendremos uno de cada cinco element
iris[c(TRUE, FALSE, FALSE, FALSE, FALSE), ]
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
## 1            5.1         3.5          1.4         0.2     setosa
## 6            5.4         3.9          1.7         0.4     setosa
## 11           5.4         3.7          1.5         0.2     setosa
## 16           5.7         4.4          1.5         0.4     setosa
## 21           5.4         3.4          1.7         0.2     setosa
## 26           5.0         3.0          1.6         0.2     setosa
## 31           4.8         3.1          1.6         0.2     setosa
## 36           5.0         3.2          1.2         0.2     setosa
## 41           5.0         3.5          1.3         0.3     setosa
## 46           4.8         3.0          1.4         0.3     setosa
## 51           7.0         3.2          4.7         1.4 versicolor
## 56           5.7         2.8          4.5         1.3 versicolor
## 61           5.0         2.0          3.5         1.0 versicolor
## 66           6.7         3.1          4.4         1.4 versicolor
## 71           5.9         3.2          4.8         1.8 versicolor
## 76           6.6         3.0          4.4         1.4 versicolor
## 81           5.5         2.4          3.8         1.1 versicolor
## 86           6.0         3.4          4.5         1.6 versicolor
```

```
## 91           5.5         2.6          4.4         1.2 versicolor
## 96           5.7         3.0          4.2         1.2 versicolor
## 101          6.3         3.3          6.0         2.5  virginica
## 106          7.6         3.0          6.6         2.1  virginica
## 111          6.5         3.2          5.1         2.0  virginica
## 116          6.4         3.2          5.3         2.3  virginica
## 121          6.9         3.2          5.7         2.3  virginica
## 126          7.2         3.2          6.0         1.8  virginica
## 131          7.4         2.8          6.1         1.9  virginica
## 136          7.7         3.0          6.1         2.3  virginica
## 141          6.7         3.1          5.6         2.4  virginica
## 146          6.7         3.0          5.2         2.3  virginica
```

8 Funciones

La instalación base de R tiene suficientes funciones para que realicemos todas las tareas básicas de análisis de datos, desde importar información hasta crear documentos para comunicarla (¡este libro ha sido creado con R!).

Sin embargo, es comun que necesitemos realizar tareas para las que no existe una función específica o que para encontrar solución necesitemos combinarl o utilizar funciones en sucesión, lo cual puede complicar nuestro código.

Ilustremos lo anterior con un ejemplo.

8.1 ¿Por qué necesitamos crear nuestrar propias funciones?

Supongamos que tenemos un jefe que nos ha pedido crear un histograma con datos de edad que hemos recogido en una encuesta.

Esto es sencillo de resolver pues contamos con la función `hist()` que hace exactamente esto. Sólo tenemos que dar un vector numérico como argumento para generar una gráfica (veremos esto con más detalle en el capítulo 12).

Primero, generaremos datos aleatorios sacados de una distribución normal con la función `rnorm()`. Esta función tiene los siguientes argumentos:

- `n`: Cantidad de números a generar.
- `mean`: Media de la distribución de la que sacaremos nuestros números.
- `sd`: Desviación estándar de la distribución de la que sacaremos nuestros números.

Además, llamaremos `set.seed()` para que estos resultados sean replicables. Cada que llamamos `rnorm()` se generan número aleatorios diferentes, pero si antes llamamos a `set.seed()`, con un número específico como argumento obtendremos los mismos resultados.

Obtendremos 1500 números con media 15 y desviación estándar .75.

```
set.seed(173)
edades <- rnorm(n = 1500, mean = 15, sd = .75)
```

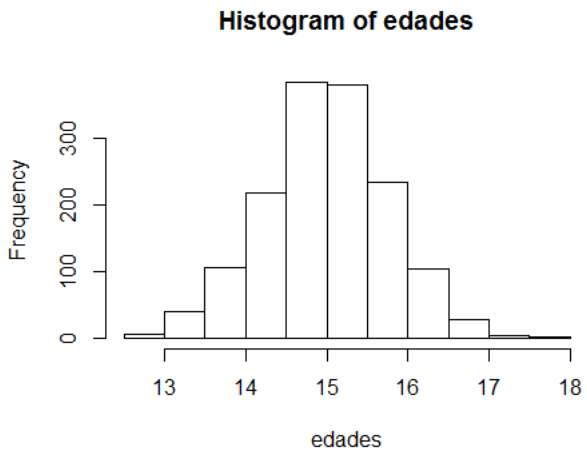
Veamos los primero diez números de nuestro objeto.

```
edades[1:10]
```

```
##  [1] 15.79043 14.68603 16.29119 14.66079 15.25658 14.62890 14.87498
##  [8] 16.35364 16.04607 16.35803
```

Ahora, sólo tenemos que ejecutar `hist()` con el argumento `x` igual a nuestro vector y obtendremos un histograma.

```
# Histograma
hist(x = edades)
```



Estupendo. Hemos logrado nuestro objetivo.

Nuestro jefe está satisfecho, pero le gustaría que en el histograma se muestre la media y desviación estándar de los datos, que tenga un título descriptivo y que los ejes estén etiquetados en español, además de que las barras sean de color dorado.

Suena complicado, pero podemos calcular la media de los datos usando la función `mean()`, la desviación estándar con `sd()` y podemos agregar los resultados de este cálculo al histograma usando la función `abline()`. Para agregar título, etiquetas en español y colores al histograma sólo basta agregar los argumentos apropiados a la función `hist()`.

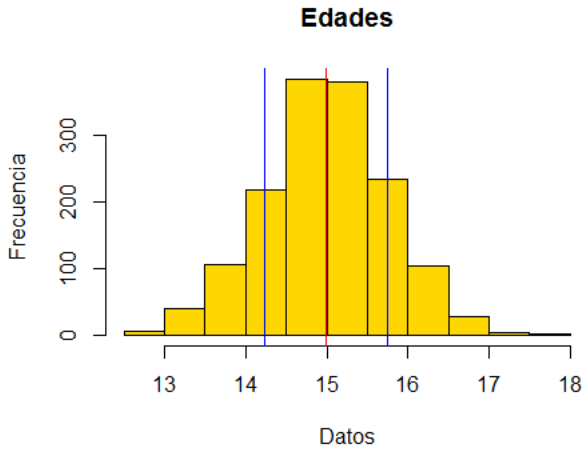
No te preocupes mucho por los detalles de todo esto, lo veremos más adelante.

Calculamos media y desviación estándar de nuestros datos.

```
media <- mean(edades)
desv_est <- sd(edades)
```

Agregamos líneas con `abline()`, para la media de rojo y desviación estándar con azúl. También ajustamos los argumentos de `hist()`.

```
hist(edades, main = "Edades", xlab = "Datos", ylab = "Frecuencia", col = "gold")
abline(v = media, col = "red")
abline(v = media + (desv_est * c(1, -1)), col = "blue")
```



Con esto nuestro jefe ahora sí ha quedado complacido. Tanto, que nos pide que hagamos un histograma igual para todas las variables numéricas de esa encuesta. Que son cincuenta en total.

Para cumplir con esta tarea podríamos usar el código que ya hemos escrito. Simplemente lo copiamos y pegamos cincuenta veces, cambiando los valores para cada una de variables que nos han pedido.

Pero hacer las cosas de este modo propicia errores y es difícil de corregir y actualizar.

Para empezar, si copias el código anterior cincuenta veces, tendrás un script con más de **400 líneas**. Si en algún momento te equivocas porque escribiste "Enceusta" en lugar de "Encuesta", incluso con las herramientas de búsqueda de RStudio, encontrar donde está el error será una tarea larga y tediosa.

Y si tu jefe en esta ejemplo quiere que agregues, quites o modifiques tu histograma, tendrás que hacer el cambio cincuenta veces, una para cada copia del código. De nuevo, con esto se incrementa el riesgo de que ocurran errores.

Es en situaciones como esta en las que se hace evidente la necesidad de crear nuestras propias funciones, capaces de realizar una tarea específica a nuestros problemas, y que pueda usarse de manera repetida. Así reducimos errores, facilitamos hacer correcciones o cambios y nos hacemos la vida más fácil, a nosotros y a quienes usen nuestro código después.

8.2 Funciones definidas por el usuario

Una función tiene un nombre, argumentos y un cuerpo. Las funciones definidas por el usuario son creadas usando la sigiente estructura.

```
nombre <- function(argumentos) {
  operaciones
}
```

Cuando asignamos una función a un nombre decimos que hemos **definido una función**.

El **nombre** que asignamos a una función nos permite ejecutarla y hacer referencias a ella. Podemos asignar la misma función a diferentes nombres o cambiar una función a la que ya le hemos asignado un nombre. Es recomendable elegir nombres claros, no ambiguos y descriptivos.

Una vez que la función tiene nombre, podemos llamarla usando su nombre, al igua lque con las funciones por defecto de R.

Los **argumentos** son las variables que necesita la función para realizar sus operaciones. Aparecen entre paréntesis, separados por comas. Los valores son asignados al nombre del argumento por el usuario cada vez que ejecuta una función. Esto permite que usemos nuestras funciones en distintas situaciones con diferentes datos y especificaciones.

Los argumentos pueden ser datos, estructuras de datos, conexiones a archivos u otras funciones y todos deben tener nombres diferentes.

El **cuerpo** de la función contiene, entre llaves, todas las operaciones que se ejecutarán cuando la función sea llamada. El cuerpo de una función puede ser tan sencillo o complejo como nosotros deseemos, incluso podemos definir funciones dentro de una función (y definir funciones dentro de una función dentro de otra función, aunque esto se vuelve confuso rápidamente).

Si el código del cuerpo de la función tiene errores, sus operaciones no se realizarán y nos será devuelto un mensaje de error **al ejecutarla**. R no avisa si nuestra función va a funcionar o no hasta que intentamos correrla.

Una ventaja de usar RStudio es que nos indica errores de sintaxis en nuestro código, lo cual puede prevenir algunos errores. Sin embargo, hay alguno que no detecta, como realizar operaciones o coerciones imposibles.

Para ver esto en acción, crearemos una función sencilla para obtener el área de un cuadrilátero.

8.3 Nuestra primera función

Partimos del algoritmo para calcular el área de un cuadrilátero: `lado x lado`.

Podemos convertir esto a operaciones de R y asignarlas a una función llamada `area_cuad` de la siguiente manera:

```
area_cuad <- function(lado1, lado2) {
  lado1 * lado2
}
```

Las partes de nuestra función son:

- **Nombre**: `area_cuad`.
- **Argumentos**: `lado1`, `lado2`. Estos son los datos que necesita la función para calcular el área, representan el largo de los lados de un cuadrilátero.
- **Cuerpo**: La operación `lado1 * lado2`, escrita de manera que R pueda interpretarla.

Ejecutaremos nuestra función para comprobar que funciona. Nota que lo único que hacemos cada que la llamamos es cambiar la medida de los lados del cuadrilátero para el que calcularemos un área, en lugar de escribir la operación `lado1 * lado2` en cada ocasión.

```
area_cuad(lado1 = 4, lado2 = 6)
```

```
## [1] 24
```

```
area_cuad(lado1 = 36, lado2 = 36)
```

```
## [1] 1296
```

En cada llamada a nuestra función estamos asignando valores distintos a los argumentos usando el signo de igual. Si no asignamos valores a un argumento, se nos mostrará un error

```
area_cuad(lado1 = 14)
```

```
## Error in area_cuad(lado1 = 14): argument "lado2" is missing, with no default
```

En R, podemos especificar los argumentos por posición. El orden de los argumentos se determina cuando creamos una función.

En este caso, nosotros determinamos que el primer argumento que recibe `area_cuad` es `lado1` y el segundo es `lado2`. Así, podemos escribir lo siguiente y obtener el resultado esperado.

```
area_cuad(128, 64)
```

```
## [1] 8192
```

Esto es equivalente a escribir `lado1 = 128, lado2 = 64` como argumentos.

Podemos crear ahora una función ligeramente más compleja para calcular el volumen de un prisma rectangular

Siguiendo la misma lógica de transformar un algoritmo a código de R, podemos crear una función con el algoritmo: `arista x arista x arista`.

Definimos la función `area_prisma()`.

```
area_prisma <- function(arista1, arista2, arista3) {
  arista1 * arista2 * arista3
}
```

Probemos nuestra función.

```
area_prisma(arista1 = 3, arista2 = 6, arista3 = 9)
```

```
## [1] 162
```

También podríamos escribir esta función aprovechando nuestra función `area_cuad`.

```
area_prisma <- function(arista1, arista2, arista3) {
  area_cuad(arista1, arista2) * arista3
}

# Probemos la función
area_prisma(3, 6, 9)
```

```
## [1] 162
```

Con esto estamos listos para definir una función para crear histogramas con las características que nos pidió nuestro jefe hipotético.

8.4 Definiendo la función `crear_histograma()`

Definiremos una función con el nombre `crear_histograma()` para generar un gráfico con las especificaciones que se nos han pedido.

Partimos de una función sin argumentos y el cuerpo vacio.

```
crear_histograma <- function() {

}
```

Para que esta función realice lo que deseamos necesitamos:

- Los datos que serán graficados.
- El nombre de la variable graficada

Por lo tanto, nuestros argumentos serán:

- `datos`
- `nombre`

```
crear_histograma <- function(datos, nombre) {

}
```

Ya sabemos las operaciones realizaremos, sólo tenemos que incluirlas al cuerpo de nuestro función.

Reemplazaremos las variables que hacen referencia a un objeto en particular por el nombre de nuestros argumentos. De esta manera será generalizable a otros casos.

En este ejemplo, cambiamos la referencia a la variable **edades** por referencias al argumento `datos` y la referencia a **Edades**, que usaremos como título del histograma, por una referencia al argumento `nombre`.

```
crear_histograma <- function(datos, nombre) {
  media <- mean(datos)
  desv_est <- sd(datos)

  hist(datos, main = nombre, xlab = "Datos", ylab = "Frecuencia", col = "gold")
  abline(v = media, col = "red")
  abline(v = media + (desv_est * c(1, -1)), col = "blue")
}
```

Probemos nuestra función usando datos distintos, generados de manera similar a las edades, con la función `rnorm()`.

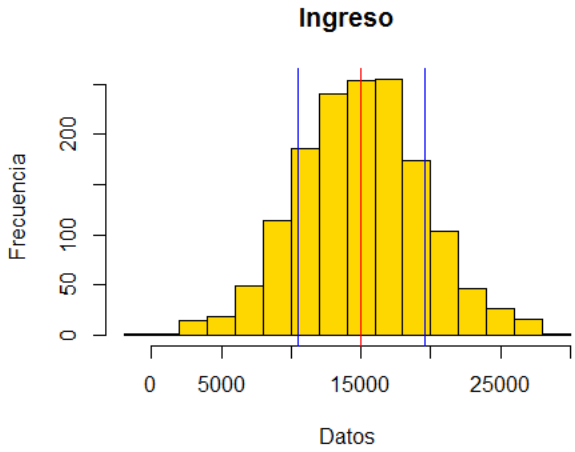
Generaremos datos de ingreso, con una media igual a 15000 y una desviación estándar de 4500.

```
ingreso <- rnorm(1500, mean = 15000, sd = 4500)

# Resultado
ingreso[1:10]
```

```
##  [1] 14365.18 16621.70 13712.35 21796.08 14226.73 13830.29 22187.37
##  [8] 17879.22 11040.41 17923.13
```

Corremos nuestra función.

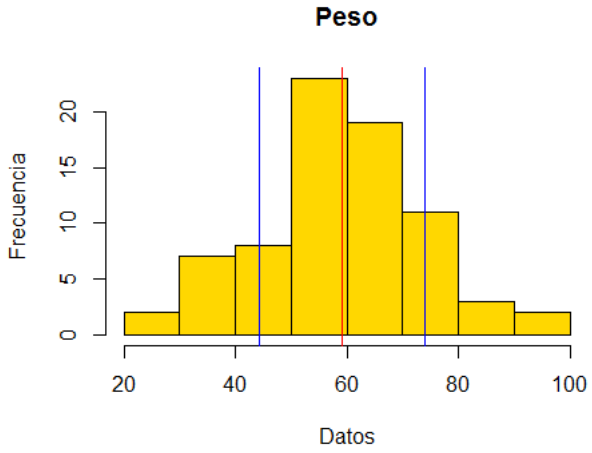
```
crear_histograma(ingreso, "Ingreso")
```



Luce bien. Probemos ahora con datos sobre el peso de las personas. siguiendo el mismo procedimiento.

```
peso <- rnorm(75, mean = 60, sd = 15)
```

```
crear_histograma(peso, "Peso")
```



Las funciones definidas por el usuario pueden devolvernos errores. Por ejemplo, si introducimos datos que no son apropiados para las operaciones a realizar, nuestra función no se ejecutará correctamente.

```
crear_histograma("Cuatro", ingreso)
```

```
## Warning in mean.default(datos): argument is not numeric or logical:
## returning NA
```

```
## Warning in var(if (is.vector(x) || is.factor(x)) x else as.double(x), na.rm
## = na.rm): NAs introduced by coercion
```

```
## Error in hist.default(datos, main = nombre, xlab = "Datos", ylab = "Frecuencia", : 'x' must be
```

Por esta razón es importante crear **documentación** para las funciones que hayas creado. Puede ser tan sencilla como una explicación de qué hace la función y qué tipo de datos necesita para realizar sus operaciones.

La primera persona beneficiada por esto eres tu, pues tu yo de un mes en el futuro puede haber olvidado por completo la lógica de una función específica, así que la documentación es una manera de recordar tu trabajo.

Una manera simple de documentar tus funciones es con comentarios.

```
# crear_histograma
# Devuelve un histograma con lineas indicando la media y desviación estándar de un vector de dat
# Argumentos:
# - datos: Un vector numérico.
# - nombre: Una cadena de texto.
```

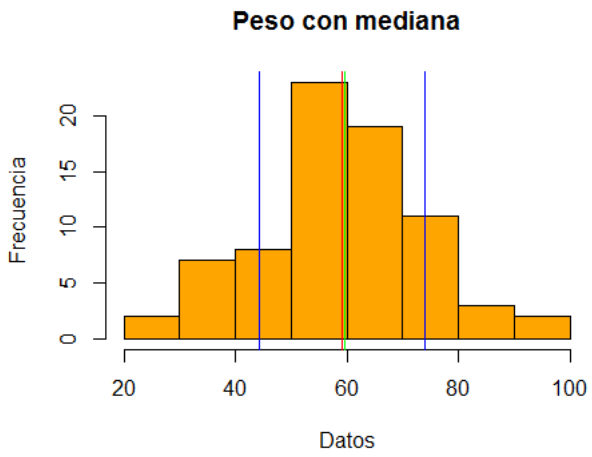
8.4.1 Ejecutando

Ahora, podremos cumplir con la solicitud de nuestro jefe ficticio usando cincuenta llamadas a una función en lugar de correr más de cuatrocientas líneas de código y que hemos reducido la probabilidad de cometer errores.

Además, si a nuestro jefe se le ocurren nuevas características para los histogramas, basta con cambiar el cuerpo de nuestra función una vez y esto se verá reflejado en nuestro cincuenta casos al correr de nuevo el código.

Por ejemplo, supongamos que nuestro jefe también quiere que el histograma muestre la mediana de nuestros datos y que las barras sean de color naranja. Basta con hacer un par de cambios.

```
crear_histograma <- function(datos, nombre) {
  media <- mean(datos)
  desv_est <- sd(datos)
  mediana <- median(datos)

  hist(datos, main = nombre, xlab = "Datos", ylab = "Frecuencia", col = "orange")
  abline(v = media, col = "red")
  abline(v = media + (desv_est * c(1, -1)), col = "blue")
  abline(v = mediana, col = "green")
}

# Resultado
crear_histograma(peso, "Peso con mediana")
```



Quizás estés pensando que escribir una función cincuenta veces de todos modos es demasiada repetición y aún se presta a cometer errores. Lo cual es cierto, pero podemos hacer más breve nuestro código y menos susceptible a equivocaciones con la familia de funciones **apply**, que revisaremos en el capítulo 10.

9 Estructuras de control

Como su nombre lo indica, las estructuras de control nos permiten controlar la manera en que se ejecuta nuestro código.

Las estructuras de control establecen **condicionales** en nuestros código. Por ejemplo, qué condiciones deben cumplirse para realizar una operación o qué debe ocurrir para ejecutar una función.

Esto es de gran utilidad para determinar la lógica y el orden en que ocurren las operaciones, en especial al definir funciones.

Las estructuras de control más usadas en R son las siguientes.

Estructura de control	Descripción
`if`, `else`	Si, de otro modo
`for`	Para cada uno en
`while`	Mientras
`break`	Interrupción
`next`	Siguiente

9.1 if, else

`if` (si) es usado cuando deseamos que una operación se ejecute únicamente cuando una condición se cumple.

`else` (de otro modo) es usado para indicarle a R qué hacer en caso de la condición de un `if` no se cumpla.

Un `if` es la manera de decirle a R:

- **SI** esta condición es cierta, **ENTONCES** haz estas operaciones.

El modelo para un `if` es:

```
if(Condición) {
  operaciones_si_la_condición_es_TRUE
}
```

Si la condición se cumple, es decir, es verdadera (`TRUE`), entonces se realizan las operaciones. En caso contrario, no ocurre nada y el código con las operaciones no es ejecutado.

Por ejemplo, le pedimos a R que nos muestre el texto "Verdadero" si la condición se cumple.

```
# Se cumple la condición y se muestra "verdadero"
if(4 > 3) {
  "Verdadero"
}
```

```
## [1] "Verdadero"
```

```
# No se cumple la condición y no pasa nada
if(4 > 5) {
  "Verdadero"
}
```

`else` complementa un `if`, pues indica qué ocurrirá cuando la condición no se cumple, es falsa (`FALSE`), en lugar de no hacer nada.

Un `if` con `else` es la manera de decirle a R:

- **SI** esta condición es es cierta, **ENTONCES** haz estas operaciones, **DE OTRO MODO** haz estas otras operaciones.

El modelo para un **if** con un **else** es:

```
if(condición) {
  operaciones_si_la_condición_es_TRUE
} else {
  operaciones_si_la_condición_es_FALSE
}
```

Usando los ejemplos anteriores, podemos mostrar "Falso" si no se cumple la condición, en lugar de que no ocurra nada.

```
# Se cumple la condición y se muestra "Verdadero"
if(4 > 3) {
  "Verdadero"
} else {
  "Falso"
}
```

```
## [1] "Verdadero"
```

```
# No se cumple la condición y se muestra "Falso"
if(4 > 5) {
  "Verdadero"
} else {
```

```
  "Falso"
}
```

```
## [1] "Falso"
```

9.1.1 Usando if y else

Para ilustrar el uso de `if else` definiremos una función que calcule el promedio de calificaciones de un estudiante y, dependiendo de la calificación calculada, nos devuelva un mensaje específico.

Empezamos definiendo una función para calcular promedio. En realidad, sólo es la aplicación de la función `mean()` ya existente en R *base*, pero la ampliaremos después.

```
promedio <- function(calificaciones) {
  mean(calificaciones)
}
```

```
promedio(c(6, 7, 8, 9, 8))
```

```
## [1] 7.6
```

```
promedio(c(5, 8, 5, 6, 5))
```

```
## [1] 5.8
```

Ahora deseamos que esta función nos muestre si un estudiante ha aprobado o no.

Si asumimos que un estudiante necesita obtener 6 o más de promedio para aprobar, podemos decir que:

- **SI** el promedio de un estudiante es igual o mayor a 6, **ENTONCES** mostrar "Aprobado", **DE OTRO MODO**, mostrar "Reprobado".

Aplicamso esta lógica con un `if`, `else` en la función `promedio()`.

```
promedio <- function(calificaciones) {
  media <- mean(calificaciones)

  if(media >= 6) {
    print("Aprobado")
  } else {
    print("Reprobado")
  }
}
```

Probemos nuestra función

```
promedio(c(6, 7, 8, 9, 8))
```

```
## [1] "Aprobado"
```

```
promedio(c(5, 8, 5, 6, 5))
```

```
## [1] "Reprobado"
```

Está funcionando, aunque los resultados podrían tener una mejor presentación.

Usaremos la función `paste0()` para pegar el promedio de calificaciones, como texto, con el resultado de "Aprobado" o "Reprobado". Esta función acepta como argumentos cadenas de texto y las pega (concatena) entre sí, devolviendo como resultado una nueva cadena.

Primero concatenaremos la palabra "Calificación: " a la media obtenida con la función `promedio()` y después el resultado de esta operación con la palabra "aprobado" o "reprobado", según corresponda.

```
promedio <- function(calificaciones) {
  media <- mean(calificaciones)
  texto <- paste0("Calificación: ", media, ", ")

  if(media >= 6) {
    print(paste0(texto, "aprobado"))
  } else {
    print(paste0(texto, "reprobado"))
  }
}
```

Pongamos a prueba nuestra función.

```
promedio(c(6, 7, 8, 9, 8))
```

```
## [1] "Calificación: 7.6, aprobado"
```

```
promedio(c(5, 8, 5, 6, 5))
```

```
## [1] "Calificación: 5.8, reprobado"
```

Por supuesto, como lo vimos en el capítulo sobre funciones, podemos hacer aún más compleja a `promedio()`, pero esto es suficiente para conocer mejor las aplicaciones de `if else`.

9.1.2 ifelse

La función `ifelse( )` nos permite vectorizar `if, else`. En lugar de escribir una línea de código para cada comparación, podemos usar una sola llamada a esta función, que se aplicará a todos los elementos de un vector.

Si intentamos usar `if else` con un vector, se nos mostrará una advertencia.

```
if(1:10 < 3) {
  "Verdadero"
}
```

```
## Warning in if (1:10 < 3) {: the condition has length > 1 and only the first
## element will be used
```

```
## [1] "Verdadero"
```

Este mensaje nos dice que sólo se usará el primer elemento del vector para evaluar su la condición es verdadera y lo demás será ignorado.

En cambio, con `ifelse` se nos devolverá un valor para cada elemento de un vector en el que la condición sea `TRUE`, además nos devolverá otro valor para los elementos en que la condición sea `FALSE`.

Esta función tiene la siguiente forma.

```
ifelse(vector, valor_si_TRUE, valor_si_FALSE)
```

Si intentamos el ejemplo anterior con `ifelse()`, se nos devolverá un resultado para cada elemento del vector, no sólo del primero de ellos.

```
##  [1] "Verdadero" "Verdadero" "Falso"     "Falso"     "Falso"
##  [6] "Falso"     "Falso"     "Falso"     "Falso"     "Falso"
```

De este modo podemos usar `ifelse()` para saber si los números en un vector son pares o nones.

```
num <- 1:8
```

```
ifelse(num %% 2 == 0, "Par", "Non")
```

```
## [1] "Non" "Par" "Non" "Par" "Non" "Par" "Non" "Par"
```

También tenemos la opción de crear condiciones más complejas usando operadores Booleanos.

Por ejemplo, pedimos sólo los números que son exactamente divisibles entre 2 y 3.

```
num <- 1:20

ifelse(num %% 2 == 0 & num %% 3, "Divisible", "No divisible")
```

```
##  [1] "No divisible" "Divisible"    "No divisible" "Divisible"
##  [5] "No divisible" "No divisible" "No divisible" "Divisible"
##  [9] "No divisible" "Divisible"    "No divisible" "No divisible"
## [13] "No divisible" "Divisible"    "No divisible" "Divisible"
## [17] "No divisible" "No divisible" "No divisible" "Divisible"
```

Desde luego, esto es particularmente útil para recodificar datos.

```
num <- c(0, 1, 0, 0, 0, 1, 1)

num <- ifelse(num == 0, "Hombre", "Mujer")

num
```

```
## [1] "Hombre" "Mujer"  "Hombre" "Hombre" "Hombre" "Mujer"  "Mujer"
```

9.2 for

La estructura `for` nos permite ejecutar un bucle (*loop*), realizando una operación para cada elemento de un conjunto de datos.

Su estructura es la siguiente:

```
for(elemento in objeto) {
  operacion_con_elemento
}
```

Con lo anterior le decimos a R:

- **PARA** cada elemento **EN** un objeto, haz la siguiente operación.

Al escribir un bucle `for` la parte que corresponde al **elemento** la podemos llamar como nosotros deseemos, pero la parte que corresponde al **objeto** debe ser el nombre de un objeto existente.

Los dos bucles siguientes son equivalentes, sólo cambia el nombre que le hemos puesto al **elemento**.

```
objeto <- 1:10

for(elemento in objeto) {
  operacion_con_elemento
}
```

```
for(i in objeto) {
  operacion_con_elemento
}
```

Tradicionalmente se usa la letra **i** para denotar al elemento, pero nosotros usaremos nombres más descriptivos en este capítulo.

9.2.1 Usando for

Vamos a obtener el cuadrado de cada uno de los elementos en un vector numérico del 1 al 6, que representa las caras de un dado.

```
dado <- 1:6

for(cara in dado) {
  dado ^ 2
}
```

Notarás que al ejecutar el código anterior parece que no ha ocurrido nada. En realidad, sí se han realizado las operaciones, pero R no ha devuelto sus resultados.

Las operaciones en un `for` se realizan pero sus resultados nunca son devueltos automáticamente, es necesario pedirlos de manera explícita.

A diferencia de otros lenguajes de programación en los que pedimos los resultados de un bucle con `return()`, en R este procedimiento sólo funciona con funciones.

Una solución para mostrar los resultados de un bucle `for` es usar la función `print()`.

```
for(cara in dado) {
  print(cara ^ 2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
```

Comprobamos que la operación ha sido realizada a cada elemento de nuestro objeto. Sin embargo, usar `print()` sólo mostrará los resultados de las operaciones en la consola, no los asignará a un objeto.

Si deseamos asignar los resultados de un bucle `for` a un objeto, usamos índices.

Aprovechamos que el primer elemento en un bucle siempre es identificado con el número **1** y que continuará realizando operaciones hasta llegar al total de elementos que hemos especificado.

```
for(numero in 1:10) {
  print(numero)
}
```

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

En nuestro ejemplo, pasamos por los valores de dado, cara por cara. La primera cara será igual a **1**, la segunda a **2**, y así sucesivamente hasta el **6**.

Podemos usar estos valores para asignar cada valor resultante de nuestras operaciones a una posición específica en un vector, incluso si este está vacio.

Creamos un vector vacío, asignándole como `NULL` como valor.

```
mi_vector <- NULL
```

Ejecutamos nuestro bucle.

```
for(cara in dado) {
  mi_vector[cara] <- cara ^ 2
}
```

Aunque no fueron mostrados en la consola, los resultados han sido asignados al objeto `mi_vector`.

```
mi_vector
```

```
## [1]  1  4  9 16 25 36
```

9.2.2 for y vectorización

Notarás que el resultado que obtuvimos usando **for** es el mismo que si vectorizamos la operación.

```
dado ^ 2
```

```
## [1]  1  4  9 16 25 36
```

Dado que en R contamos con vectorización de operaciones, que podemos usar las funciones de la familia apply (discutido en siguiente capítulo) en objetos diferentes a vectores y que la manera de recuperar los resultados de un `for` es un tanto laboriosa, este tipo de bucle no es muy popular en R.

En R generalmente hay opciones mejores, en cuanto a simplicidad y velocidad de cómputo, que un bucle `for`.

Sin embargo, es conveniente que conozcas esta estructura de control, pues hay ocasiones en la que es la mejor herramienta para algunos problemas específicos.

9.3 while

Este es un tipo de bucle que ocurre **mientras** una condición es verdadera (`TRUE`). La operación se realiza hasta que se se llega a cumplir un criterio previamente establecido.

El modelo de **while** es:

```
while(condicion) {
  operaciones
}
```

Con esto le decimos a R:

- **MIENTRAS** esta condición sea **VERDADERA**, haz estas operaciones.

La condición generalmente es expresada como el resultado de una o varias operaciones de comparación, pero también puede ser el resultado de una función.

9.3.1 Usando while

Probemos sumar `+1` a un valor, mientras que este sea menor que 5. Al igual que con `for`, necesitamos la función `print()` para mostrar los resultados en la consola.

```
umbral <- 5
valor <- 0
```

```
while(valor < umbral) {
  print("Todavía no.")
  valor <- valor + 1
}
```

```
## [1] "Todavía no."
## [1] "Todavía no."
## [1] "Todavía no."
## [1] "Todavía no."
## [1] "Todavía no."
```

¡Ten cuidado con crear bucles infinitos! Si ejecutas un `while` con una condición que nunca será `FALSE`, este nunca se detendrá.

Si corres lo siguiente, presiona la tecla **ESC** para detener la ejecución, de otro modo, correra por siempre y puede llegar a congelar tu equipo.

```
while(1 < 2) {
  print("Presiona ESC para detener")
}
```

El siguiente es un error común. Estamos sumando 1 a `i` con cada iteración del bucle, pero como no estamos asignando este nuevo valor á `i`, su valor se mantiene igual, entonces la condición nunca se cumplirá y el bucle será infinito.

De nuevo, si corres lo siguiente, presiona la tecla **ESC** para detener la ejecución.

```
i <- 0
while(i < 10) {
  i + 1
}
```

Un uso común de **while** es que realice operaciones que queremos detener cuando se cumple una condición, pero desconocemos cuándo ocurrirá est

Supongamos que, por alguna razón queremos sumar calificaciones, del 1 al 10 al azar, hasta llegar a un número que mayor o igual a 50. Además nos interesa saber cuántas calificaciones sumaron y cuál fue el resultado al momento de cumplir la condición.

Para obtener números al azar del 1 al 10, usamos la función `sample()`. Esta función va a tomar una muestra al azar de tamaño igual a 1 (argumento `size`) de un vector del 1 al 10 (argumento `x`) cada vez que se ejecute.

Por lo tanto, cada vez que corras el ejemplo siguiente obtendrás un resultado distinto, pero siempre llegarás a un valor mayor a 50.

Creamos dos objetos, `conteo` y `valor`. Les asignamos el valor 0.

```
conteo <-  0
valor <- 0
```

Nuestro `while` hará dos cosas.

Primero, tomará un número al azar del 1 al 10, y lo sumará a `valor`. Segundo, le sumará 1 a `conteo` cada que esto ocurra, de esta manera sabremos cuántas iteraciones ocurrieron para llegar a un valor que no sea menor a 50.

```
while(valor < 50) {
  valor <- valor + sample(x = 1:10, size = 1)
  conteo <- conteo + 1
}
```

Aunque no son mostrados en la consola los resultados son asignados a los objetos `valor`y `conteo`

```
valor
```

```
## [1] 52
```

```
conteo
```

```
## [1] 8
```

Por último, si intentamos ejecutar un `while` para el que la condición nunca es igual a `TRUE`, este no realizará ninguna operación.

```
conteo <- 0

while("dado" == "ficha") {
  conteo <- conteo + 1
}

conteo
```

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

9.4 break y next

`break` y `next` son **palabras reservadas** en R, no podemos asignarles nuevos valores y realizan una operación específica cuando aparecen en nuestro código.

`break` nos permite **interrumpir** un bucle, mientras que `next` nos deja avanzar a la **siguiente** iteración del bucle, "saltándose" la actual. Ambas funcionan para `for` y `while`.

9.4.1 Usando break

Para interrumpir un bucle con `break`, necesitamos que se cumpla una condición. Cuando esto ocurre, el bucle se detiene, aunque existan elementos a los cuales aún podría aplicarse.

Interrumpimos un `for` cuando `i` es igual a 3, aunque aún queden 7 elementos en el objeto.

```
for(i in 1:10) {
  if(i == 3) {
    break
  }
  print(i)
}
```

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

Interrumpimos un `while` antes de se cumpla la condición de que `numero` sea mayor a 5, en cuanto este tiene el valor de 15.

```
numero <- 20

while(numero > 5) {
  if(numero == 15) {
    break
  }
  numero <- numero - 1
}

numero
```

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

Como habrás notado, la aplicación de `break` es muy similar a `while`, realizar una operación hasta que se cumple una condición, y ambos pueden usarse en conjunto.

9.4.2 Usando next

Por su parte, usamos next para "saltarnos" una iteración en un bucle. Cuando la condición se cumple, esa iteración es omitida.

```
for(i in 1:4) {
  if(i == 3) {
    next
  }
```

```
  print(i)
}
```

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

Estas dos estructuras de control nos dan un control fino sobre nuestro código. aunque los dos ejemplos de arriba son con **for**, también funcionan con **while** y **repeat**.

En realidad, **break** es indispensable para **repeat**.

9.5 repeat

Este es un bucle que se llevará a cabo el número de veces que especifiquemos, usando un `break` para detenerse. `repeat` asegura que las operaciones que contiene sean iteradas al menos en una ocasión.

La estructura de **repeat** es el siguiente:

```
repeat {
  operaciones

  un_break_para_detener
}
```

Si no incluimos un `break`, el bucle se repetirá indefinidamente y sólo lo podremos detener pulsando la tecla ESC, así que hay que tener cuidado al usar esta estructura de control.

Por ejemplo, el siguiente `repeat` sumará `+1` a `valor` hasta que este sea igual a cinco, entonces se detendrá.

```
valor <-  0
mi_vector <- NULL

repeat{
  valor <- valor + 1
  if(valor == 5) {
    break
  }
}

# Resultado
valor
```

```
## [1] 5
```

Este tipo de bucle es quizás el menos utilizado de todos, pues en R existen alternativas para obtener los mismos resultados de manera más sencilla y sin el riesgo de crear un bucle infinito. Sin embargo, puede ser la mejor alternativa para problemas específicos.

10 La familia apply

La familia de funciones `apply` es usada para aplicar una función a cada elemento de una estructura de datos. En particular, es usada para aplicar funciones en matrices, data frames, arrays y listas.

Con esta familia de funciones podemos automatizar tareas complejas usando poca líneas de código y es una de las características distintivas de R como lenguaje de programación.

La familia de funciones `apply` es una expresión de los rasgos del paradigma funcional de programación presentes en R. Sobre esto no profundizaremos demasiado, pero se refiere saber que en R las funciones son "ciudadanos de primera", con la misma importancia que los objetos, y por lo tanto, operamos en ellas.

La familia de funciones apply no sólo recibe datos como argumentos, también recibe funciones.

10.0.1 Un recordatorio sobre vectorización

Para entender más fácilmente el uso de la familia apply, recordemos la vectorización de operaciones.

Hay operaciones que, si las aplicamos a un vector, son aplicadas a todos sus elementos.

```
mi_vector <- 1:10
```

```
mi_vector
```

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

```
mi_vector ^ 2
```

```
##  [1]   1   4   9  16  25  36  49  64  81 100
```

Lo anterior es, generalmente, preferible a escribir una operación para cada elemento o a usar un bucle **for**, como se describió en el capítulo sobre estructuras de control.

Como todo lo que ocurre en R es una función, podemos decir que **al vectorizar estamos aplicando una función a cada elemento de un vector**. La familia de funciones **apply** nos permite implementar esto en estructuras de datos distintas a los vectores.

10.0.2 Las funciones de la familia apply

La familia apply esta formada por las siguientes funciones:

- `apply()`
- `eapply()`
- `lapply()`
- `mapply()`
- `rapply()`
- `sapply()`
- `tapply()`
- `vapply()`

Es una familia numerosa y esta variedad de funciones se debe a que varias de ellas tienen aplicaciones sumamente específicas.

Todas las funciones de esta familia tienen una característica en común: **reciben como argumentos a un objeto y al menos una función**.

Hasta ahora, todas las funciones que hemos usado han recibido como argumentos estructuras de datos, sean vectores, data frames o de otro tipo. Las funciones de la familia apply tienen la particularidad que pueden recibir a otra función como un argumento. Lo anterior puede sonar confuso, pero es más bien intuitivo al verlo implementado.

Nosotros trabajaremos con las funciones más generales y de uso común de esta familia:

- `apply()`
- `lapply()`

Estas dos funciones nos permitirán solucionar casi todos los problemas a los que nos encontremos. Además, conociendo su uso, las demás funciones de la familia **apply** serán relativamente fáciles de entender.

10.1 apply

`apply` aplica una función a todos los elementos de una **matriz**.

La estructura de esta función es la siguiente.

```
apply(X, MARGIN, FUN)
```

`apply` tiene tres argumentos:

- `X`: Una matriz o un objeto que pueda coercionarse a una matriz, generalmente, un data frame.
- `MARGIN`: La dimensión (margen) que agrupará los elementos de la matriz `X`, para aplicarles una función. Son identificadas con números, **1** son renglones y **2** son columnnas.
- `FUN`: La función que aplicaremos a la matriz `X` en su dimención `MARGIN`.

10.1.1 ¿Qué es X

`X` es una matriz o cualquier otro objeto que sea posible coercionar a una matriz. Esto es, principalmente, vectores y data frames.

Recuerda que puedes coercionar objetos a matriz usando `as.matrix()` y puedes comprobar si un objeto es de esta clase con `is.matrix()`.

```
# Creamos un data frame
mi_df <- data.frame(v1 = 1:3, v2 = 4:6)

mi_df
```

```
##   v1 v2
## 1  1  4
## 2  2  5
## 3  3  6
```

```
# Coerción a matriz
mi_matriz <- as.matrix(mi_df)

# Verificamos que sea matriz
is.matrix(mi_matriz)
```

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

```
# Resultado
mi_matriz
```

```
##      v1 v2
## [1,]  1  4
## [2,]  2  5
## [3,]  3  6
```

Aunque también podemos coercionar listas y arrays a matrices, los resultados que obtenemos no siempre son apropiados para `apply()`, por lo que no es recomendable usarl estos objetos como argumentos.

10.1.2 ¿Qué es MARGIN?

Recuerda que las matrices y los data frames están formadas por vectores y que estas estructuras tienen dos dimensiones, ordenadas en renglones y columnas. Esto lo vimos en en Matrices y arrays y Data frames.

Para `MARGIN`:

- 1 es renglones.
- 2 es columnas.

Por ejemplo, podemos usar `apply()` para obtener la sumatoria de los elementos de una matriz, por renglón.

Creamos una matriz de cuatro renglones.

```
matriz <- matrix(1:14, nrow = 4)
```

```
## Warning in matrix(1:14, nrow = 4): data length [14] is not a sub-multiple
## or multiple of the number of rows [4]
```

Aplicamos `apply()`, dando la función `sum()` el argumento `FUN`, nota que sólo necesitamos el nombre de la función, sin paréntesis.

Por último, damos el argumento `MARGIN = 1`, para aplicar la función por renglón.

```
apply(X = matriz, MARGIN = 1, FUN = sum)
```

```
## [1] 28 32 22 26
```

Esto es equivalente a hacer lo siguiente.

```
sum(matriz[1, ])
```

```
## [1] 28
```

```
sum(matriz[2, ])
```

```
## [1] 32
```

```
sum(matriz[3, ])
```

```
## [1] 22
```

```
sum(matriz[4, ])
```

```
## [1] 26
```

Y naturalmente, es equivalente a hacer lo siguiente.

```
sum(vector_1)
```

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

```
sum(vector_2)
```

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

```
sum(vector_3)
```

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

```
sum(vector_4)
```

```
## [1] 58
```

Estamos aplicando una función a cada elemento de nuestra matriz. Los elementos son los renglones. Cada renglón es un vector. Cada vector es usado como argumento de la función.

Si cambiamos el argumento MARGIN de `MARGIN = 1` a `MARGIN = 2`, entonces la función se aplicará por columna.

```
apply(X = matriz, MARGIN = 2, FUN = sum)
```

```
## [1] 10 26 42 30
```

En este caso, la función `sum()` ha sido aplicado a cada elementos de nuestra matriz, los elementos son las columnas, y cada columna es un vector.

10.1.3 ¿Qué es FUN?

FUN es un argumento que nos pide el **nombre de una función que se se aplicarla a todos los elementos de nuestra matriz**.

El ejemplo de la sección anterior aplicamos las funciones `mean()` y `sum()` usando sus nombres, sin paréntesis, esto es, sin especificar argumentos.

Podemos dar como argumento cualquier nombre de función, siempre y cuando ésta acepte vectores como argumentos.

Probemos cambiando el argumento `FUN`. Usaremos la función `mean()` para obtener la media de cada renglón y de cada columna.

Aplicado a los renglones.

```
apply(matriz, 1, mean)
```

```
## [1] 7.0 8.0 5.5 6.5
```

Aplicado a las columnas

```
apply(matriz, 2, mean)
```

```
## [1]  2.5  6.5 10.5  7.5
```

Las siguientes llamadas a `sd()`, `max()` y `quantile()` se ejecutan sin necesidad de especificar argumentos.

```
# Desviación estándar
apply(matriz, 1, FUN = sd)
```

```
## [1] 5.163978 5.163978 4.434712 4.434712
```

```
# Máximo
apply(matriz, 1, FUN = max)
```

```
## [1] 13 14 11 12
```

```
# Cuantiles
apply(matriz, 1, FUN = quantile)
```

```
##      [,1] [,2] [,3] [,4]
## 0%      1    2  1.0  2.0
## 25%     4    5  2.5  3.5
## 50%     7    8  5.0  6.0
## 75%    10   11  8.0  9.0
## 100%   13   14 11.0 12.0
```

10.1.4 ¿Cómo sabe FUN cuáles son sus argumentos?

Recuerda que podemos llamar una función y proporcionar sus argumentos en orden, tal como fueron establecidos en su definición.

Por lo tanto, **el primer argumento que espera la función, será la `x` del `apply()`**.

Para ilustrar esto, usaremos la función `quantile()`. Llama `?quantile` en la consola para ver su documentación.

```
?quantile
```

`quantile()` espera siempre un argumento `x`, que debe ser un vector numérico, además tener varios argumentos adicionales.

- `probs` es un vector numérico con las probabilidades de las que queremos extraer cuantiles.
- `na.rm`, si le asignamos `TRUE` quitará de x los `NA` y `NaN` antes de realizar operaciones.
- `names`, si le asignamos `TRUE`, hará que el objeto resultado de la función tenga nombres.
- `type` espera un valor entre 1 y 9, para determinar el algoritmo usado para el cálculo de los cuantiles.

En orden, el primer argumento es `x`, el segundo `probs`, y así sucesivamente.

Cuando usamos `quantile()` en un `apply()`, el argumento `x` de la función será cada elemento de nuestra matriz. Es decir, los vectores como renglones o columnas de los que está constituida la matriz.

Esto funcionará siempre y cuando los argumentos sean apropiados para la función. Si proporcionamos un argumento inválido, la función no se ejecutará y **apply** fallará.

Por ejemplo, intentamos obtener cuantiles de las columnas de una matriz, en la que una de ellas es de tipo caracter.

Creamos una matriz.

```
matriz2 <- matrix(c(1:2, "a", "b"), nrow = 2)

# Resultado
```

Aplicamos la función y obtenemos un error.

```
apply(matriz2, 2, quantile)
```

```
## Error in (1 - h) * qs[i]: non-numeric argument to binary operator
```

Por lo tanto, **apply sólo puede ser usado con funciones que esperan vectores como argumentos**.

10.1.5 ¿Qué pasa si deseamos utilizar los demás argumentos de una función con apply?

En los casos en los que una función tiene recibe más de un argumento, asignamos los valores de estos del nombre de la función, separados por comas, usando sus propios nombres (a este procedimiento es al que se refiere el argumento `...` descrito en la documentación de `apply`).

Supongamos que deseamos encontrar los cuantiles de un vector, correspondientes a las probabilidades **.33** y **.66**. Esto es definido con el argumento `probs` de esta función.

Para ello, usamos `quantile()` y después de haber escrito el nombre de la función, escribimos el nombre del argumento probs y los valores que deseamos para este.

```
apply(X = matriz, MARGIN = 2, FUN = quantile, probs = c(.33, .66))
```

```
##     [,1] [,2]  [,3]  [,4]
## 33% 1.99 5.99  9.99  1.99
## 66% 2.98 6.98 10.98 12.78
```

Como podrás ver, hemos obtenido los resultados esperados.

Si además deseamos que el resultado aparezca sin nombres, entonces definimos el valor del argumento `names` de la misma manera.

```
apply(matriz, 2, quantile, probs = c(.33, .66), names = FALSE)
```

```
##      [,1] [,2]  [,3]  [,4]
## [1,] 1.99 5.99  9.99  1.99
## [2,] 2.98 6.98 10.98 12.78
```

De este modo es posible aplicar funciones complejas que aceptan múltiples argumentos, con la ventaja que usamos pocas líneas de código.

10.1.6 ¿Qué tipo de resultados devuelve apply?

En los ejemplos anteriores, el resultado de `apply()` en algunas ocasiones fue un vector y en otros fue una matriz.

Si aplicamos `mean()`, obtenemos como resultado un vector.

```
mat_media <- apply(matriz, 1, mean)

class(mat_media)

## [1] "numeric"
```

Pero si aplicamos `quantile()`, obtenemos una matriz.

```
mat_cuant <- apply(matriz, 1, quantile)

class(mat_cuant)

## [1] "matrix"
```

Este comportamiento se debe a que **`apply()` nos devolverá objetos del mismo tipo que la función aplicada devuelve**. Dependiendo de la función, será el tipo de objeto que obtengamos.

Sin embargo, este comportamiento puede causarte algunos problemas. En primer lugar, anterior te obliga a conocer de antemano el tipo del resultado que obtendrás, lo cual no siempre es fácil de determinar, en particular si las funciones que estás utilizando son poco comunes o tienen comportamientos poco convencionales.

Cuando estás trabajando en proyectos en los que el resultado de una operación será usado en operaciones posteriores, corres el riesgo de que en alguna parte del proceso, un `apply()` te devuelva un resultado que te impida continuar adelante.

Con algo de práctica es más o menos sencillo identificar problemas posibles con los resultados de `apply()`, pero es algo que debes tener en cuenta, pues puede explicar por qué tu código no funciona como esperabas.

En este sentido, `lapply()` tiene la ventaja de que siempre devuelve una lista.

10.2 lapply

`lapply()` es un caso especial de `apply()`, diseñado para **aplicar funciones a todos los elementos de una lista**. La **l** de su nombre se refiere, precisamente, a **lista**.

`lapply()` intentará coercionar a una lista el objeto que demos como argumento y después aplicará una función a todos sus elementos.

`lapply` siempre nos devolverá una lista como resultado. A diferencia de `apply`, sabemos que siempre obtendremos ub objeto de tipo lista después de aplicar una función, sin importar cuál función sea.

Dado que en R todas las estructuras de datos pueden coercionarse a una lista, `lapply()` puede usarse en un número más amplio de casos que `apply()`, además de que esto nos permite utilizar funciones que aceptan argumentos distintos a vectores.

La estructura de esta función es:

```
lapply(X, FUN)
```

En donde:

- `X` es una lista o un objeto coercionable a una lista.
- `FUN` es la función a aplicar.

Estos argumentos son idéntico a los de `apply()`, pero a diferencia aquí no especificamos `MARGIN`, pues las listas son estructuras con una unidimensionales, que sólo tienen largo.

10.2.1 Usando lapply()

Probemos `lapply()` aplicando una función a un data frame. Usaremos el conjunto de datos `trees`, incluido por defecto en R *base*.

`trees` contiene datos sobre el grueso, alto y volumen de distinto árboles de cerezo negro. Cada una de estas variables está almacenada en una columna del data frame.

Veamos los primeros cinco renglones de `trees`.

```
trees[1:5, ]
```

```
##   Girth Height Volume
## 1   8.3     70   10.3
## 2   8.6     65   10.3
## 3   8.8     63   10.2
## 4  10.5     72   16.4
## 5  10.7     81   18.8
```

Aplicamos la función `mean()`, usando su nombre.

```
lapply(X = trees, FUN = mean)
```

```
## $Girth
## [1] 13.24839
##
## $Height
## [1] 76
##
## $Volume
## [1] 30.17097
```

Dado que un data frame está formado por columnas y cada columna es un vector atómico, cuando usamos `lapply()` , la función es aplicada a cada columna. `lapply()`, a diferencia de `apply()` no puede aplicarse a renglones.

En este ejemplo, obtuvimos la media de grueso (Girth), alto (Height) y volumen (Volume), como una lista.

Verificamos que la clase de nuestro resultado es una lista con `class()`.

```
arboles <- lapply(X = trees, FUN = mean)

class(arboles)
```

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

Esto es muy conveniente, pues la recomendación para almacenar datos en un data frame es que cada columna represente una variable y cada renglón un caso (por ejemplo, el enfoque **tidy** de Wickham (2014)). Por lo tanto, con `lapply()` podemos manipular y transformar datos, por variable.

Al igual que con `apply()`, podemos definir argumentos adicionales a las funciones que usemos, usando sus nombres, después del nombre de la función.

```
lapply(X = trees, FUN = quantile, probs = .8)
```

```
## $Girth
##  80%
## 16.3
##
## $Height
## 80%
##  81
##
## $Volume
##  80%
## 42.6
```

Si usamos `lapply` con una matriz, la función se aplicará a cada **celda** de la matriz, no a cada columna.

Creamos una matriz.

```
matriz <- matrix(1:9, ncol = 3)

# Resultado
matriz
```

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

Llamamos a `lapply()`.

```
lapply(matriz, quantile, probs = .8)
```

```
## [[1]]
## 80%
##   1
##
## [[2]]
## 80%
##   2
##
## [[3]]
## 80%
##   3
##
## [[4]]
## 80%
##   4
##
## [[5]]
## 80%
##   5
##
## [[6]]
## 80%
##   6
##
## [[7]]
## 80%
##   7
##
## [[8]]
## 80%
##   8
##
## [[9]]
## 80%
##   9
```

Para usar una matriz con `lapply()` y que la función se aplique a cada columna, primero la coercionamos a un data frame con la función `as.data.frame()`

```
lapply(as.data.frame(matriz), quantile, probs = .8)
```

```
## $V1
## 80%
## 2.6
##
## $V2
## 80%
## 5.6
##
## $V3
## 80%
## 8.6
```

Si deseamos aplicar una función a los renglones de una matriz, una manera de lograr es transponer la matriz con `t()` y después coercionar a un data frame.

```
matriz_t <- t(matriz)

lapply(as.data.frame(matriz_t), quantile, probs = .8)
```

```
## $V1
## 80%
## 5.8
##
## $V2
## 80%
## 6.8
##
## $V3
## 80%
## 7.8
```

Con vectores como argumento, `lapply()` aplicará la función a cada elementos del vector, de manera similar a una vectorización de operaciones.

Por ejemplo, usamos `lapply()` para obtener la raíz cuadrada de un vector numérico del 1 al 4, con la función `sqrt()`.

```
mi_vector <- 1:4

lapply(mi_vector, sqrt)
```

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

```
##
## [[2]]
## [1] 1.414214
##
## [[3]]
## [1] 1.732051
##
## [[4]]
## [1] 2
```

10.2.2 Usando lapply() en lugar de un bucle for

En muchos casos es posible reemplazar un bucle `for()` por un `lapply()`.

De hecho, `lapply()` está haciendo lo mismo que un `for()`, está iterando una operación en todos los elementos de una estructura de datos.

Por lo tanto, el siguiente código con un `for()`...

```
mi_vector <- 6:12
resultado <- NULL
posicion <- 1

for(numero in mi_vector) {
  resultado[posicion] <- sqrt(numero)
  posicion <- posicion + 1
}

resultado
```

```
## [1] 2.449490 2.645751 2.828427 3.000000 3.162278 3.316625 3.464102
```

... nos dará los mismos resultados que el siguiente código con `lapply()`.

```
resultado <- NULL

resultado <- lapply(mi_vector, sqrt)

resultado
```

```
## [[1]]
## [1] 2.44949
##
## [[2]]
## [1] 2.645751
##
## [[3]]
## [1] 2.828427
##
## [[4]]
```

```
## [1] 3
##
## [[5]]
## [1] 3.162278
##
## [[6]]
## [1] 3.316625
##
## [[7]]
## [1] 3.464102
```

El código con `lapply()` es mucho más breve y más sencillo de entender, al menos para otros usuarios de R.

El inconveniente es que obtenemos una lista como resultado en lugar de un vector, pero eso es fácil de resolver usando la función `as.numeric()` para hacer coerción a tipo numérico.

```
as.numeric(resultado)
```

```
## [1] 2.449490 2.645751 2.828427 3.000000 3.162278 3.316625 3.464102
```

El siguiente código es la manera en la que usamos `for()` si deseamos aplicar una función a todas sus columnas, tiene algunas partes que no hemos discutido, pero es sólo para ilustrar la diferencia simplemente usar `trees_max <- lapply(trees, max)`.

```
trees_max <- NULL
i <- 1
columnas <- ncol(trees)

for(i in 1:columnas) {
  trees_max[i] <- max(trees[, i])
  i <- i +1
}

trees_max
```

```
## [1] 20.6 87.0 77.0
```

10.2.3 Usando lapply con listas

Hasta hora hemos hablado de usar `lapply()` con objetos que pueden coercionarse a una lista, pero ¿qué pasa si usamos esta función con una lista que contiene a otros objetos?

Pues la función se aplicará a cada uno de ellos. Por lo tanto, así podemos utilizar funciones que acepten todo tipo de objetos como argumento. Incluso podemos aplicar funciones a listas recursivas, es decir, listas de listas.

Por ejemplo, obtendremos el coeficiente de correlación de cuatro data frames contenidos en una sola lista. Esto no es posible con `apply()`, porque sólo podemos usar funciones que aceptan vectores como argumentos, pero con `lapply()` no es ningún problema.

Empezaremos creando una lista de data frames. Para esto, usaremos las función `rnorm()`, que genera números al azar y `set.seed()`, para que obtengas los mismos resultados aquí mostrados.

`rnorm()` creara `n` números al azar (pseudoaleatorios, en realidad), sacados de una distribución normal con media 0 y desviación estandar 1. `set.seed()` es una función que "fija" los resultados de una generación de valores al azar. Cada que ejecutas `rnorm()` obtienes resultados diferentes, pero si das un número como argumento `seed` a `set.seed()`, siempre obtendrás los mismos números.

```
# Fijamos seed
set.seed(seed = 2018)

# Creamos una lista con tres data frames dentro
tablas <- list(
  df1 = data.frame(a = rnorm(n = 5), b = rnorm(n = 5), c = rnorm(n = 5)),
  df2 = data.frame(d = rnorm(n = 5), e = rnorm(n = 5), f = rnorm(n = 5)),
  df3 = data.frame(g = rnorm(n = 5), h = rnorm(n = 5), i = rnorm(n = 5))
)


# Resultado
tablas
```

```
## $df1
##             a          b          c
## 1 -0.42298398 -0.2647112 -0.6430347
## 2 -1.54987816  2.0994707 -1.0300287
## 3 -0.06442932  0.8633512  0.7124813
## 4  0.27088135 -0.6105871 -0.4457721
## 5  1.73528367  0.6370556  0.2489796
##
## $df2
##            d          e          f
## 1 -1.0741940  1.2638637 -0.2401222
## 2 -1.8272617  0.2501979 -1.0586618
## 3  0.0154919  0.2581954  0.4194091
## 4 -1.6843613  1.7855342 -0.2709566
## 5  0.2044675 -1.2197058 -0.6318248
##
## $df3
##            g          h          i
## 1 -0.2284119 -0.4897908 -0.3594423
## 2  1.1786797  1.4105216 -1.2995363
## 3 -0.2662727 -1.0752636 -0.8698701
## 4  0.5281408  0.2923947  1.0543623
## 5 -1.7686592 -0.2066645 -0.1486396
```

Para obtener el coeficiente de correlación usaremos la función `cor()`.

Esta función acepta como argumento una data frame o una matriz. Con este objeto, calculará el coeficiente de correlación **R de Pearson** existente entre cada una de sus columnas. Como resultado obtendremos una matriz de correlación.

Por ejemplo, este es el resultado de aplicar `cor()` a `iris`.

```
cor(iris[1:4])
```

```
##              Sepal.Length Sepal.Width Petal.Length Petal.Width
## Sepal.Length    1.0000000  -0.1175698    0.8717538   0.8179411
## Sepal.Width    -0.1175698   1.0000000   -0.4284401  -0.3661259
## Petal.Length    0.8717538  -0.4284401    1.0000000   0.9628654
## Petal.Width     0.8179411  -0.3661259    0.9628654   1.0000000
```

Con `lapply` aplicaremos `cor()` a cada uno de los data frames contenidos en nuestra lista. El resultado será una lista de matrices de correlaciones.

Esto lo logramos con una línea de código.

```
lapply(X = tablas, FUN = cor)
```

```
## $df1
##            a          b          c
## a  1.0000000 -0.4427336  0.6355358
## b -0.4427336  1.0000000 -0.1057007
## c  0.6355358 -0.1057007  1.0000000
##
## $df2
##            d          e         f
## d  1.0000000 -0.6960942 0.4709283
## e -0.6960942  1.0000000 0.2624429
## f  0.4709283  0.2624429 1.0000000
##
## $df3
##            g          h          i
## g  1.0000000  0.6228793 -0.1472657
## h  0.6228793  1.0000000 -0.1211321
## i -0.1472657 -0.1211321  1.0000000
```

De esta manera puedes manipular información de múltiples data frames, matrices o listas con muy pocas líneas de código y, en muchos casos, más rápidamente que con las alternativas existentes.

Finalmente, si asignamos los resultados de las última operación a un objeto, podemos usarlos y manipularlos de la misma manera que cualquier otra lista.

```r
correlaciones <- lapply(tablas, cor)

# Extraemos el primer elemento de la lista
correlaciones[[1]]
```

```
##            a          b          c
## a  1.0000000 -0.4427336  0.6355358
## b -0.4427336  1.0000000 -0.1057007
## c  0.6355358 -0.1057007  1.0000000
```

11 Importar y exportar datos

Hasta ahora, hemos trabajado con datos ya existentes en R *base* o que hemos generado nosotros mismos, sin embargo, lo usual es que usemos datos almacenados en archivos fuera de R.

R puede importar datos de una amplia variedad de tipos de archivo con las funciones en *base* además de que esta capacidad es ampliada con el uso de paquetes específicos.

Cuando importamos un archivo, estamos guardando su contenido en nuestra sesión como un objeto. Dependiendo del procedimiento que usemos será el tipo de objeto creado.

De manera análoga, podemos exportar nuestros objetos de R a archivos en nuestra computadora.

11.1 Descargando datos

Antes de empezar a importar datos, vale la pena señalar que podemos descargar archivos de internet usando R con la función `download.file()`.

De esta manera tendremos acceso a una vasta diversidad de fuentes de datos. Entre otras, podrás descargar los archivos

La función `download.file()` nos pide como argumento `url`, la dirección de internet del archivo que queremos descargar y `destfile` el nbombre que tendrá el archivo en nuestra computadora. Ambos argumentos como cadenas de texto, es decir, entre comillas.

Por ejemplo, para descargar una copia del set *iris* disponible en el *UCI Machine Learning Repository* usamos la siguiente dirección como argumento `url`:

- https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data

Y asignamos "iris.data" al argumento `dest`.

```
download.file(
  url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data",
  destfile = "iris.data"
  )
```

El resultado es un archivo llamado "iris.data" en nuestro directorio de trabajo.

Este método funciona con prácticamente todo tipo de archivos, aunque en algunos casos será necesario agregar el argumento `method = "wb"`, por asegurar que el archivo obtenido funcione correctamente.

11.2 Tablas (datos rectangulares)

Como vimos en el capítulo 7, las estructura rectangular, en renglones y columnas, es común y conveniente para el análisis de datos. Nos referiremos a esta forma de organizar datos como **tabla**.

R cuenta con la función genérica `read.table()`, que puede leer cualquier tipo de archivo que contenga una tabla.

La condición para que R interprete un archivo como una tabla es que tenga renglones y en cada renglon, los datos estén separados por comas, o algún otro caracter, indicando columnas. Es decir, algo que luzca de la siguiente manera.

1, 20, 8, 5

1, 31, 6, 5

2, 18, 9, 5

2, 25, 10, 5

Por supuesto, en lugar de comas podemos tener puntos y coma, dos puntos, tabuladores o cualquier otro signo de puntuación como **separador** de columnas.

La función `read.table()` acepta un número considerable de argumentos. Los más importantes son los siguientes.

- `file`: La ruta del archivo que importaremos, como cadena de texto. Si el archivo se encuentra en nuestro directorio de trabajo, es suficientedar el nombre del archivo, sin la ruta completa.
- `header`: Si nuestro archivo tiene encabezados, para ser interpretados como nombres de columna, definimos este argumento como `TRUE`.
- `sep`: El caracter que es usado como separador de columnas. Por defecto es ";".
- `col.names`: Un vector opcional, de tipo caracter, con los nombres de las columnas en la tabla.
- `stringsAsFactors`: Esta función convierte automáticamente los datos de texto a factores. Si este no es el comportamiento que deseamos, definimos este argumento como `FALSE`.

Puedes consultar todos los argumentos de esta función ejecutando `?read.table` en la consola.

Es importante señalar que el objeto obtenido al usar esta función es siempre un **data frame**.

Probemos con un archivo con extensión ".data", descargado desde el respositorio de Github de este libro.

```
download.file(
  url = "https://raw.githubusercontent.com/jboscomendoza/r-principiantes-bookdown/master/datos/b
  dest = "breast-cancer-wis.data"
)
```

Estos datos pertenecen a una base de diagnósticos de cancer mamario de la Universidad de Wisconsin, usado para probar métodos de aprendizaje automático. Puedes encontrar la información completa sobre este conjunto de datos en el siguiente enlace:

- https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29

Nos damos cuenta de que hemos tenido éxito en la descarga si aparece un mensaje en la consola de R indicando los resultados de nuesta operación.

Usamos sin especificar ningún otro argumento.

```
bcancer <- read.table(file = "breast-cancer-wis.data")
```

Veamos los primeros renglones de nuestros datos usando la función `head()`

```
head(bcancer)
```

```
##                              V1
## 1    1000025,5,1,1,1,2,1,3,1,1,2
## 2   1002945,5,4,4,5,7,10,3,2,1,2
## 3    1015425,3,1,1,1,2,2,3,1,1,2
## 4    1016277,6,8,8,1,3,4,3,7,1,2
## 5    1017023,4,1,1,3,2,1,3,1,1,2
## 6 1017122,8,10,10,8,7,10,9,7,1,4
```

Nuestros datos no lucen particularmente bien. Necesitamos ajustar algunos parámetros al importarlos.

No hay datos de encabezado, por lo que `header` será igual a `FALSE` y el separador de columnas es una coma, así que el valor de `sep` será ",". No conocemos cuál es el nombre de las columnas, así que por el momento no proporcionaremos uno.

```
bcancer <- read.table(file = "breast-cancer-wis.data", header = FALSE, sep = ",")

# Resultado
head(bcancer)
```

```
##        V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11
## 1 1000025  5  1  1  1  2  1  3  1   1   2
## 2 1002945  5  4  4  5  7 10  3  2   1   2
## 3 1015425  3  1  1  1  2  2  3  1   1   2
## 4 1016277  6  8  8  1  3  4  3  7   1   2
## 5 1017023  4  1  1  3  2  1  3  1   1   2
## 6 1017122  8 10 10  8  7 10  9  7   1   4
```

Luce mejor, pero los nombres de las columnas son poco descriptivos. Si no damos nombres de variables, cada columna tendrá como nombre "V" seguida de números del 1 adelante.

Para este ejemplo, contamos con un archivo de información, que describe el contenido de los datos que hemos importado.

- https://raw.githubusercontent.com/jboscomendoza/r-principiantes-bookdown/master/datos/breast-cancer-wis.names

Si descargas este archivo, puedes abrirlo usando el bloc o navegador de internet de tu computadora.

Guardaremos en un vector las abreviaturas de los nombres de columna descritos en el documento anterior.

```
nombres <- c("id", "clump_t", "u_csize", "u_cshape", "m_adh", "spcs", "b_nuc",
             "b_chr", "n_nuc", "mit", "class")
```

Ahora usaremos este vector como argumento `col.names` en `read.table()`, para importar nuestros datos con nombres de columna.

```
bcancer <- read.table(file = "breast-cancer-wis.data", header = FALSE, sep = ",",
                      col.names = nombres)

# Resultado
head(bcancer)
```

```
##        id clump_t u_csize u_cshape m_adh spcs b_nuc b_chr n_nuc mit class
## 1 1000025       5       1        1     1    2     1     3     1   1     2
## 2 1002945       5       4        4     5    7    10     3     2   1     2
## 3 1015425       3       1        1     1    2     2     3     1   1     2
## 4 1016277       6       8        8     1    3     4     3     7   1     2
## 5 1017023       4       1        1     3    2     1     3     1   1     2
## 6 1017122       8      10       10     8    7    10     9     7   1     4
```

Nuestros datos han sido importados correctamente. Además, el objeto resultante es un data frame, listo para que trabajemos con él.

```
class(bcancer)
```

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

11.2.1 Archivos CSV

Un caso particular de las tablas, son los archivos separados por comas, con extensión **.csv**, por *Comma Separated Values*, sus siglas en inglés. Este es un tipo de archivo comunmente usado para compartir datos, pues es compatible con una amplia variedad de sistemas diferentes además de que ocupa relativamente poco espacio de almacenamiento.

Este tipo de archivos también se pueden importar usando la función `read.table()`.

Probemos descargando los mismos datos que en el ejemplo anterior, pero almacenados en un archivo con extensión **.csv**.

```
download.file(
  url = "https://raw.githubusercontent.com/jboscomendoza/r-principiantes-bookdown/master/datos/b
  dest = "breast-cancer-wis.csv"
)
```

Podemos usar `read.table()` con los mismos argumentos que en el ejemplo anterior, con la excepción de que este archivo sí tiene encabezados de columna, por lo que cambiamos `header` de `FALSE` a `TRUE`.

```
bcancer <- read.table(file = "breast-cancer-wis.csv", header = TRUE, sep = ",",
                      col.names = nombres)

# Resultado
head(bcancer)
```

```
##        id clump_t u_csize u_cshape m_adh spcs b_nuc b_chr n_nuc mit class
## 1 1000025       5       1        1     1    2     1     3     1   1     2
## 2 1002945       5       4        4     5    7    10     3     2   1     2
## 3 1015425       3       1        1     1    2     2     3     1   1     2
## 4 1016277       6       8        8     1    3     4     3     7   1     2
## 5 1017023       4       1        1     3    2     1     3     1   1     2
## 6 1017122       8      10       10     8    7    10     9     7   1     4
```

Una ventaja de usar documentos con extensión **.csv** es la posibilidad de usar la función `read.csv()`. Esta es una es una versión de `read.table()`, optimizada para importar archivos **.csv**.

`read.csv()` acepta los mismos argumentos que `read.table()`, pero al usarla con un archivo **.csv**, en casi todo los casos, no hara falta especificar nada salvo la ruta del archivo.

```
bcancer <- read.csv("breast-cancer-wis.csv")

# Resultado
head(bcancer)
```

```
##        id clump_t u_csize u_cshape m_adh spcs b_nuc b_chr n_nuc mit class
## 1 1000025       5       1        1     1    2     1     3     1   1     2
## 2 1002945       5       4        4     5    7    10     3     2   1     2
## 3 1015425       3       1        1     1    2     2     3     1   1     2
## 4 1016277       6       8        8     1    3     4     3     7   1     2
## 5 1017023       4       1        1     3    2     1     3     1   1     2
## 6 1017122       8      10       10     8    7    10     9     7   1     4
```

`read.csv()` también devuelve un data frame como resultado

11.3 Archivos con una estructura desconocida

Habrá ocasiones en las que no estamos seguros del contenido de los archivos que deseamos importar. En estos casos, podemos pedirle a R que intente abrir el archivo en cuestión, usando la función `file.show()`.

Por ejemplo, intentamos abrir el archivo con extensión **.csv** que importamos antes.

```
file.show("breast-cancer-wis.csv")
```

R intentará usar el programa que en nuestro equipo, por defecto, abre el tipo de archivo que le hemos indicado. Si no tenemos un programa configurado para abrir el tipo de archivo que deseamos, nuestro sistema operativo nos pedira que elijamos uno.

Lo anterior puede ocurrir si intentas abrir el archivo con extensión **.data** que hemos importado en este capítulo.

```
file.show("breast-cancer-wis.data")
```

Podemos usar la función `readLines()` para leer un archivo línea por línea. Establecemos el argumento `n = 4` para obtener sólo los primeros cuatro renglones del documento.

```
readLines("breast-cancer-wis.data", n = 4)
```

```
## [1] "1000025,5,1,1,1,2,1,3,1,1,2"  "1002945,5,4,4,5,7,10,3,2,1,2"
## [3] "1015425,3,1,1,1,2,2,3,1,1,2"  "1016277,6,8,8,1,3,4,3,7,1,2"
```

La salida es una lista de vectores, uno por linea en el archivo.

Observando la salida de `readLines()` podremos determinar si el archivo que nos interesa puede ser importado usando con los métodos que hemos revisado o necesitaremos de herramientas diferentes.

El documento "R Data Import/Export" (R Core Team, 2018) contiene una guía avanzada sobre el proceso de importar y exportar todo tipo de datos. Puedes consultarlo en el siguiente enlace:

- https://cran.r-project.org/doc/manuals/r-release/R-data.pdf

11.4 Exportar datos

Un paso muy importante en el trabajo con R es exportar los datos que hemos generado, ya sea para que sean usados por otras personas o para almacenar información en nuestro disco duro en lugar de nuestro RAM.

Dependiendo del tipo de estructura de dato en el que se encuentran contenidos nuestros datos son las opciones que tenemos para exportarlos.

11.4.1 Data frames y matrices

Si nuestros datos se encuentran contenidos en una estructura de datos rectangular, podemos exportarlos con diferentes funciones.

De manera análoga a `read.table()`, la función `write.table()` nos permite exportar matrices o data frames, como archivos de texto con distintas extensiones.

Los argumentos más usados de `write.table()` son los siguientes.

- `x`: El nombre del data frame o matriz a exportar.
- `file`: El nombre, extensión y ruta del archivo creado con esta función. Si sólo escribimos el nombre del archivo, este será creado en nuestro directorio de tabajo.

- `sep`: El caracter que se usará como separador de columnas.
- `row.names`: Si deseamos incluir el nombre de los renglones en nuestro objeto al exportarlo, establecemos este argumento como `TRUE`. En general, es recomendable fijarlo como `FALSE`, para conservar una estructura tabular más fácil de leer.
- `col.names`: Si deseamos que el archivo incluya los nombres de las columnas en nuestro objeto, establecemos este argumento como `TRUE`. Es recomendable fijarlo como `TRUE` para evitar la necesidad de almacenar los nombres de columna en documentos distintos.

Puedes consultar todos los argumentos de esta función ejecutando `?write.table`.

Probemos exportando el objeto `iris` a un documento de texto llamado **iris.txt** a nuestro directorio de trabajo, usando como separador la coma, con nombres de columnas y sin nombre de renglones.

```
write.table(x = iris, file = "iris.txt", sep = ",",
            row.names = FALSE, col.names = TRUE)
```

Importemos el archivo que hemos creado usando `read.table()`.

```
iris_txt <- read.table(file = "iris.txt", header = TRUE, sep = ",")

# Resultado
head(iris_txt)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

También podemos exportar datos a archivos con extensión **.csv** con la función `write.csv()`.

Vamos a exportar `iris` como un documento **.csv**. En este caso, sólo especificamos que no deseamos guardar los nombres de los renglones con `row.names = FALSE`.

```
write.csv(x = iris, file = "iris.csv", row.names = FALSE)
```

Importamos el archivo creado.

```
iris_csv <- read.csv("iris.csv")

# Resultado
head(iris_csv)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

11.4.2 Listas

La manera más sencilla de exportar listas es guardarlas en archivos RDS. Este es un tipo de archivo nativo de R que puede almacenar cualquier objeto a un archivo en nuestro disco duro.

Además, RDS comprime los datos que almacena, por lo que ocupa menos espacio en disco duro que otros tipos de archivos, aunque contengan la misma información.

Para exportar un objeto a un archivo RDS, usamos la función `saveRDS()` que siempre nos pide dos argumentos:

- `object`: El nombre del objeto a exportar.
- `file`: El nombre y ruta del archivo que crearemos. Los archivos deben tener la extensión **.rds**. Si no especificamos una ruta completa, el archivo será creado en nuestro directorio de trabajo.

Creamos una lista de ejemplo que contiene dos vectores y dos matrices

```
mi_lista <- list("a" = c(TRUE, FALSE, TRUE),
     "b" = c("a", "b", "c"),
     "c" = matrix(1:4, ncol = 2),
     "d" = matrix(1:6, ncol = 3))


# Resultado
mi_lista
```

```
## $a
## [1]  TRUE FALSE  TRUE
##
## $b
## [1] "a" "b" "c"
##
## $c
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $d
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

Aunque podemos intentar `write.table()` para exportar listas, por lo general obtendremos un error como resultado.

Tratamos de exportar la lista anterior como un archivo **.txt**.

```
write.table(x = mi_lista, file = "mi_lista.txt")
```

```
## Error in (function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE, : arguments
```

Usamos la función `saveRDS()` para exportar al archivo **mi_lista.rds**.

```
saveRDS(object = mi_lista, file = "mi_lista.rds")
```

Si deseamos importar un archivo RDS a R, usamos la función `readRDS()`, indicando la ruta en la que se encuentra el archivo que deseamos.

Intentemos importar el archivo **mi_lista.rds**.

```
mi_lista_importado <- readRDS(file = "mi_lista.rds")
```

Vamos el resultado.

```
mi_lista_importado
```

```
## $a
## [1]  TRUE FALSE  TRUE
##
## $b
## [1] "a" "b" "c"
##
## $c
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $d
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
# El resultado es una lista, al igual que el objeto original
class(mi_lista)
```

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

Los objetos immportados usando un archivo RDS conservan los tipos y clases que tenían originalmente, lo cual previene pérdida de información.

11.5 Hojas de cálculo de Excel

Un formato usado con mucha frecuencia para almacenar archivos son las hojas de cálculo, en particular las generadas por el paquete *Microsoft Excel*.

R *base* no tiene una función para importar archivos almacenados en archivos con extensión **.xsl** y **.xslx**, creados con *Excel*.

Para importar datos desde este tipo de archivos, necesitamos instalar el paquete **readxl**, que contiene funciones específicas para realizar esta tarea.

Usamos la función `installpackages()`, como lo vimos en el capítulo 3

```
install.packages("readxl")
```

Ya instalado, cargamos el **readxl** a nuestra sesión de trabajo.

```
library(readxl)
```

Usaremos, principalmente dos funciones de este paquete.

- `read_excel()`: Para importar archivos **.xls** y **xlsx**.
- `excel_sheets()`: Para obtener los nombres de las pestañas en una hoja de cálculo de *Excel*.

Para probar estas funciones, descargaremos una hoja de cálculo de prueba. Nota que hemos establecido el argumento `mode = "wb"` para asegurar que el archivo se descargue correctamente.

```
download.file(
  url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/data_frames.xlsx",
  destfile = "data_frames.xlsx",
  mode = "wb"
)
```

Si intentamos leer las primeras cinco líneas de **data_frames.xlsx**, confirmamos que este es un archivo que no tiene forma rectangular, de tabla.

```
readLines("data_frames.xlsx", n = 5)
```

```
## Warning in readLines("data_frames.xlsx", n = 5): line 1 appears to contain
## an embedded nul
```

```
## Warning in readLines("data_frames.xlsx", n = 5): incomplete final line
## found on 'data_frames.xlsx'
```

```
## [1] "PK\003\004\024"
## [2] "\177ß‰YTU,B õ'(±çmöL\177¸jêd\t\001\215³¹èf\035'\200-œ6v–‹¯É{ú$\022$eµª\235…\\¬\001Åpp\177
```

En caso de que tengamos instalado *Excel* o algún otro programa compatible con archivos de hoja de cálculo, como *LibreOffice Calc* o *Number*, podemos pedir a R que abra este archivo con `file.show()`. De este modo podemos explorar su contenido.

```
file.show("data_frames.xlsx")
```

La función `excel_sheets()` nos devuelve el nombre de las pestañas como un vector.

```
excel_sheets("data_frames.xlsx")
```

```
## [1] "iris"  "trees"
```

Este archivo tiene dos pestañas, llamadas **iris** y **trees**.

Intentaremos importar la pestaña **iris** con `read_excel()`. Esta función tiene los siguientes argumentos principales.

- `path`: La ruta del archivo a importar. Si no especificamos una ruta completa, será buscado en nuestro directorio de trabajo.
- `sheet`: El nombre de la pestaña a importar. Si no especificamos este argumento, `read_excel()` intentará leer la primera pestaña de la hoja de cálculo.
- `range`: Cadena de texto con el rango de celdas a importar, escrito con el formato usado en *Excel*. Por ejemplo, "A1:B:10".
- `col_names`: Con este argumento indicamos si la pestaña que vamos a importar tiene encabezados para usar como nombres de columna. Por defecto su valor es `TRUE`. Si no tenemos encabezados, podemos dar un vector con nombres para asignar a las columnas.

Puedes consultar todos los argumentos de esta función ejecutando `?read_excel`.

Probemos `read_excel()`.

```
iris_excel <- read_excel(path = "data_frames.xlsx", sheet = "iris")
```

Nuestro resultado es un data frame.

```
iris_excel
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <chr>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
```

```
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ... with 140 more rows
```

Si los datos en la hoja de cálculo tienen forma de tabla, `read_excel()` no tendrá problemas para importarlos. Cuando este no es el caso, usamos el argumento `range` para extraer sólo la información que nos interesa.

Intentamos importar la pestaña **trees**.

```
trees_excel <- read_excel(path = "data_frames.xlsx", sheet = "trees")

# Resultado
trees_excel
```

```
## # A tibble: 34 x 6
##    `Datos trees`  X__1  X__2  X__3 X__4  X__5
##    <chr>         <dbl> <dbl> <dbl> <lgl> <chr>
##  1 <NA>           NA      NA  NA   NA    <NA>
##  2 <NA>            8.3    70  10.3 NA    <NA>
##  3 <NA>            8.6    65  10.3 NA    <NA>
##  4 <NA>            8.8    63  10.2 NA    Los nombres de las variables son~
##  5 <NA>           10.5    72  16.4 NA    <NA>
##  6 <NA>           10.7    81  18.8 NA    <NA>
##  7 <NA>           10.8    83  19.7 NA    <NA>
##  8 <NA>           11      66  15.6 NA    <NA>
##  9 <NA>           11      75  18.2 NA    <NA>
## 10 <NA>           11.1    80  22.6 NA    <NA>
## # ... with 24 more rows
```

Los resultados no lucen bien porque los datos en la pestaña no tienen forma de tabla.

Ajustamos los argumentos de `read_excel()` para leer correctamente la infomación de la pestaña. Al explorar manualmente el archivo **data.frames.xlsx**, podemos localizar el rango en el que se encuentran los datos (de las celdas B3 a D33) y los nombres de las columnas (Girth, Height y Volume).

Probemos importar de nuevo con esta información.

```
trees_excel <- read_excel(path = "data_frames.xlsx", sheet = "trees",
                          range = "B3:D33",
                          col_names = c("Girth", "Height", "Volume"))

# Resultado
trees_excel
```

```
## # A tibble: 31 x 3
##    Girth Height Volume
##    <dbl>  <dbl>  <dbl>
##  1   8.3     70   10.3
##  2   8.6     65   10.3
##  3   8.8     63   10.2
##  4  10.5     72   16.4
##  5  10.7     81   18.8
##  6  10.8     83   19.7
##  7  11       66   15.6
##  8  11       75   18.2
##  9  11.1     80   22.6
## 10  11.2     75   19.9
## # ... with 21 more rows
```

Esta vez hemos tenido éxito y los datos importados son los correctos.

El paquete **readxl** tiene más funciones para trabajar con hojas de cálculo además de `read_excel()` y `excel_sheets()`, pero revisar cada una de ellas sale del alcance de este libro. Puedes conocer más sobre ellas en la documentación de **readxl**, llamando `help(package = "readxl")`.

11.6 Datos de paquetes estadísticos comerciales (SPSS, SAS y STATA)

En ciertas disciplinas, el uso de determinados paquetes estadísticos comerciales es sumamente común. Si

Por ejemplo, en Psicología el paquete *SPSS Statistics* de IBM es el paquete estadístico comercial más usado. Si eres psicólogo o psicóloga, o colaboras con psicólogos, es áltamente probable que te encuentres con datos contenidos en archivos con extensión **.sav**, el tipo de archivo nativo de **SPSS Statistics**.

Por lo tanto, es conveniente ser capaces de importar y exportar datos almacenados en archivos compatibles con paquetes estadísticos comerciales, pues esto nos permitirá usar datos ya existentes compatibles con ellos y colaborar con otras personas.

Para este fin, usamos el paquete **haven**.

```
install.packages("haven")
```

Para usar las funciones de **haven**, lo cargamos a nuestra sesión de trabajo.

```
library(haven)
```

Las siguientes funciones de **haven** son usadas para importar datos. Todas estas funciones nos piden como argumento `file` la ruta y nombre del archivo a importar, si no especificamos ruta, será buscado en nuestro directorio de trabajo.

- `read_spss()`: *SPSS Statistics*, archivos con extensión **sav**, **zsav** y **por**.
- `read_sav()`: *SPSS Statistics*, sólo archivos **sav**, **zsav**.
- `read_sas()`: *SAS*, archivos **sas7bdat**.
- `read_xpt`: *SAS*, archivos **xpt**.
- `read_stata()`: *Stata*, archivos **dta**.

Todas importan los datos como un data frame.

También podemos exportar nuestros data frames creados en R como archivos compatibles con estos programas con las siguientes funciones. Todas piden el argumento `file`, con la ruta y nombre del archivo a crear. Es muy importante que demos como nombre de archivo uno con la extensión correcta para cada paquete.

- write_sav(): *SPSS Statistics*, archivos **sav**, **zsav** o **por**.
- write_sas(): *SAS*, archivos **sas7bda**.
- write_xpt(): *SAS*, archivos **xpt**.
- write_dta(): *Stata*, archivos **dta**.

Como siempre, puedes leer sobre las demás funciones en el paquete **haven** en su documentación, llamando `help(package = "haven")`.

12 Gráficas

R cuenta con un sistema de generación de gráficas poderoso y flexible. Si nembargo, tener estar cualidades hace que este sistema sea un tanto complejo para aprender.

En este capítulo revisaremos como crear las gráficas más comunes con R *base*, así como algunos de los parámetros que podemos ajustar para mejorar su presentación.

Al crear gráficas, notarás que ponemos en práctica todo lo que hemos visto en los capítulos anteriores, incluyendo importar datos, hacer subconjuntos de un objeto y uso de funciones.

12.1 Datos usados en el capítulo

Para las siguientes secciones utilizaremos de nuevo una copia de los datos disponibles en el *UCI Machine Learning Repository*.

Usaremos un conjunto de datos llamado *"Bank Marketing Data Set"*, que contiene información de personas contactadas en una campaña de *marketing* directo puesta en marcha por un banco de Portugal.

Comenzamos con la descarga de la copia del archivo **csv** desde el sitio de *Github* de este libro.

```
download.file(
  url = "https://raw.githubusercontent.com/jboscomendoza/r-principiantes-bookdown/master/datos/b
  destfile = "bank.csv"
  )
```

Damos un vistazo al contenido del archivo **bank.csv** con `readLines()`.

```
readLines("bank.csv", n = 4)
```

```
## [1] "\"age\";\"job\";\"marital\";\"education\";\"default\";\"balance\";\"housing\";\"loan\";\"
## [2] "30;\"unemployed\";\"married\";\"primary\";\"no\";1787;\"no\";\"no\";\"cellular\";19;\"oct
## [3] "33;\"services\";\"married\";\"secondary\";\"no\";4789;\"yes\";\"yes\";\"cellular\";11;\"m
## [4] "35;\"management\";\"single\";\"tertiary\";\"no\";1350;\"yes\";\"no\";\"cellular\";16;\"ap
```

Por la estructura de los datos, podremos usar la función `read.csv()`, con el argumento `sep = ";"` para importarlos como un data frame.

```
banco <- read.csv(file = "bank.csv", sep = ";")
```

Vemos las primeras líneas del conjunto con `head()`, el número de renglones y columnas con `dim()`.

```
# Primeros datos
head(banco)
```

```
##   age         job marital education default balance housing loan  contact
## 1  30  unemployed married   primary      no    1787      no   no cellular
## 2  33    services married secondary      no    4789     yes  yes cellular
## 3  35  management  single  tertiary      no    1350     yes   no cellular
## 4  30  management married  tertiary      no    1476     yes  yes  unknown
## 5  59 blue-collar married secondary      no       0     yes   no  unknown
## 6  35  management  single  tertiary      no     747      no   no cellular
##   day month duration campaign pdays previous poutcome  y
```

```
## 1  19   oct       79        1    -1        0  unknown no
## 2  11   may      220        1   339        4  failure no
## 3  16   apr      185        1   330        1  failure no
## 4   3   jun      199        4    -1        0  unknown no
## 5   5   may      226        1    -1        0  unknown no
## 6  23   feb      141        2   176        3  failure no
```

```
# Dimensiones
dim(banco)
```

```
## [1] 4521   17
```

Usamos `lapply()` con la función `class()` para determinar el tipo de dato de cada columna en `banco`. Conocer esto nos será muy útil más adelante.

```
lapply(banco, class)
```

```
## $age
## [1] "integer"
##
## $job
## [1] "factor"
##
## $marital
## [1] "factor"
##
## $education
## [1] "factor"
##
## $default
## [1] "factor"
##
## $balance
## [1] "integer"
##
## $housing
## [1] "factor"
##
## $loan
## [1] "factor"
##
## $contact
## [1] "factor"
##
## $day
## [1] "integer"
##
## $month
## [1] "factor"
##
## $duration
## [1] "integer"
##
## $campaign
## [1] "integer"
##
## $pdays
## [1] "integer"
##
## $previous
## [1] "integer"
##
## $poutcome
## [1] "factor"
##
## $y
## [1] "factor"
```

Y por último, pedimos un resumen de nuestros datos con la función `summary()`. Esta función acepta cualquier tipo de objeto como argumento y nos devuelve un resumen descriptivo de los datos de cada uno de sus elementos.

```
summary(banco)
```

```
##       age                 job          marital         education
##  Min.   :19.00   management :969   divorced: 528   primary  : 678
##  1st Qu.:33.00   blue-collar:946   married :2797   secondary:2306
##  Median :39.00   technician :768   single  :1196   tertiary :1350
##  Mean   :41.17   admin.     :478                   unknown  : 187
##  3rd Qu.:49.00   services   :417
##  Max.   :87.00   retired    :230
##                  (Other)    :713
##  default       balance      housing     loan           contact
##  no :4445   Min.   :-3313   no :1962   no :3830   cellular :2896
##  yes:  76   1st Qu.:   69   yes:2559   yes: 691   telephone: 301
##             Median :  444                         unknown  :1324
##             Mean   : 1423
##             3rd Qu.: 1480
##             Max.   :71188
##
##       day            month         duration       campaign
##  Min.   : 1.00   may    :1398   Min.   :   4   Min.   : 1.000
##  1st Qu.: 9.00   jul    : 706   1st Qu.: 104   1st Qu.: 1.000
##  Median :16.00   aug    : 633   Median : 185   Median : 2.000
##  Mean   :15.92   jun    : 531   Mean   : 264   Mean   : 2.794
##  3rd Qu.:21.00   nov    : 389   3rd Qu.: 329   3rd Qu.: 3.000
##  Max.   :31.00   apr    : 293   Max.   :3025   Max.   :50.000
##                  (Other): 571
##      pdays           previous          poutcome      y
##  Min.   : -1.00   Min.   : 0.0000   failure: 490   no :4000
##  1st Qu.: -1.00   1st Qu.: 0.0000   other  : 197   yes: 521
##  Median : -1.00   Median : 0.0000   success: 129
##  Mean   : 39.77   Mean   : 0.5426   unknown:3705
##  3rd Qu.: -1.00   3rd Qu.: 0.0000
##  Max.   :871.00   Max.   :25.0000
##
```

12.2 La función `plot()`

En R, la función `plot()` es usada de manera general para crear gráficos.

Esta función tiene un comportamiento especial, pues dependiendo del tipo de dato que le demos como argumento, generará diferentes tipos de gráfica. Además, para cada tipo de gráfico, podremos ajustar diferentes parámetros que controlan su aspecto, dentro de esta misma función.

Puedes imaginar a `plot()` como una especie de navaja Suiza multifuncional, con una herramienta para cada ocasión.

`plot()` siempre pide un argumento `x`, que corresponde al **eje X** de una gráfica. `x` requiere un vector y si no especificamos este argumento, obtendremos un error y no se creará una gráfica.

El resto de los argumentos de `plot()` son opcionales, pero el más importante es `y`. Este argumento también requiere un vector y corresponde al **eje Y** de nuestra gráfica.

Dependiendo del tipo de dato que demos a `x` y `y` será el gráfico que obtendremos, de acuerdo a las siguientes reglas:

`x`	`y`	**Gráfico**
Continuo	Continuo	Diagrama de dispersión (*Scatterplot*)
Continuo	Discreto	Diagrama de dispersión, `y` coercionada a numérica
Continuo	Ninguno	Diagrama de dispersión, por número de renglón

x	y	Gráfico
Discreto	Continuo	Diagrama de caja (*Box plot*)
Discreto	Discreto	Gráfico de mosaico (Diagrama de Kinneman)
Discreto	Ninguno	Gráfica de barras
Ninguno	Cualquiera	Error

En donde los tipos de dato son:

- **Continuo**: Un vector numérico, entero, lógico o complejo.
- **Discreto**: Un vector de factores o cadenas de texto.

Además de `plot()`, hay funciones que generan tipos específicos de gráfica. Por ejemplo, podemos crear una gráfica de barras con `plot()` pero existe también la función `barplot()`. También existen también casos como el de los histogramas, que sólo pueden ser creados con la función `hist()`.

Cuando llamas a la función `plot()` o alguna otra similar, R abre una ventana mostrando ese gráfico. Si estás usando RStudio, el gráfico aparece en el panel **Plot**. Si llamas de nuevo la función `plot()`, el gráfico generado más reciente reemplazará al más antiguo y en RStudio se creará una nueva pestaña en en el planel **Plot**. El gráfico reemplazado se perderá.

Por lo tanto, a menos que nosotros los indiquemos, nuestros gráficos se pierden al crear uno nuevo. Al final de este capítulo veremos cómo exportar gráficos de manera más permanente.

12.3 Histogramas

Un histograma es una gráfica que nos permite observar la distribución de datos numéricos usando barras. Cada barra representa el número de veces (frecuencia) que se observaron datos en un rango determinado.

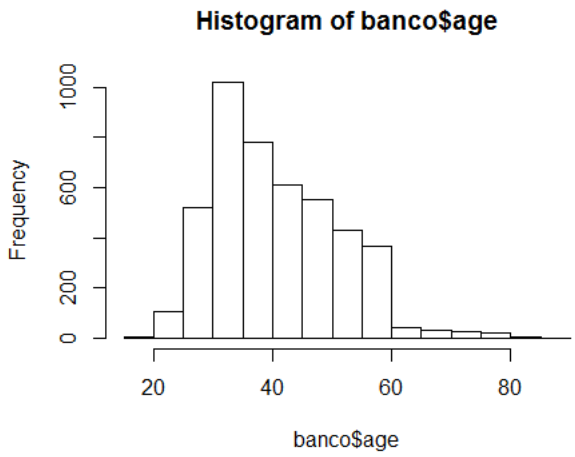
Para crear un histograma usamos la función `hist()`, que siempre nos pide como argumento `x` un vector numérico. El resto de los argumentos de esta función son opcionales. Si damos un vector no numérico, se nos devolverá un error.

Ya hemos trabajado con esta función en el capítulo 8, pero ahora profundizaremos sobre ella.

Probemos creando un histograma con las edades (*age*) de las personas en nuestro data frame `banco`. Sabemos que *age*

Daremos como argumento a `hist()` la columna **age** como un vector, extraido de `banco` usando el signo de dolar `$`, aunque también podemos usar corchetes e índices.

```
hist(x = banco$age)
```



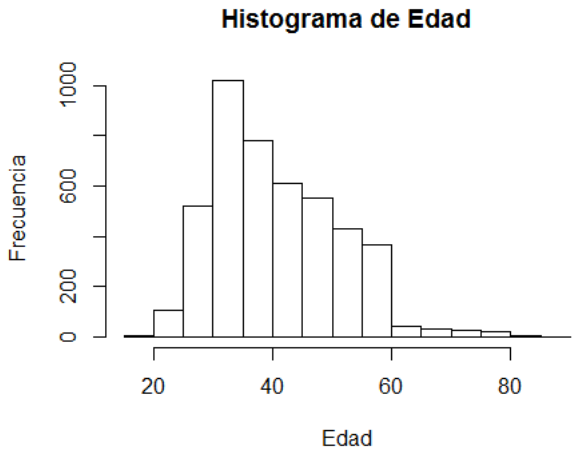
Nuestro histograma luce bastante bien para habernos costado tan poco trabajo crearlo, aunque puede mejorar su presentación.

Podemos agregar algunos argumentos a la función `hist()` para modificar ciertos parámetros gráficos.

Vamos a cambiar el título del gráfico con el argumento `main`, y el nombre de los ejes X y Y con `xlab` y `ylab`, respectivamente.

Estos argumentos requiere una cadena de texto y pueden agregados también a gráficos generados con `plot()`.

```
hist(x = banco$age, main = "Histograma de Edad",
     xlab = "Edad", ylab = "Frecuencia")
```



Probemos cambiando el color de las barras del histograma agregando el argumento `col`. Este argumento acepta nombres de colores genéricos en inglés como "red", "blue" o "purple"; y también acepta colores hexadecimales, como "#00FFFF", "#08001a" o "#1c48b5".

Puedes ver una lista de los nombres de colores válidos en R en el siguiente enlace:

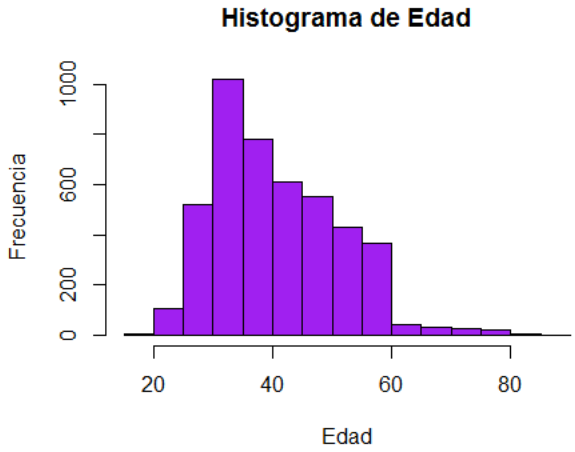
- http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf

El tema de los colores hexadecimales sale del alcance de este libro, pero en el siguiente enlace encontrarás una *web app* para generar y elegir fácilmente colores de este tipo.

- https://www.w3schools.com/colors/colors_picker.asp

Probemos con columnas de color púrpura ("purple").

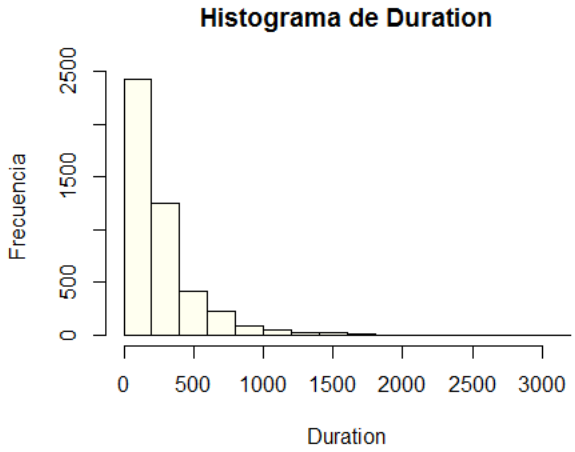
```
hist(x = banco$age, main = "Histograma de Edad",
     xlab = "Edad", ylab = "Frecuencia",
     col = "purple")
```



Nuestro histograma ya luce presentable.

Creamos ahora un histograma con los mismos argumentos, pero con los datos de la columna "duration", con barras de color marfil ("ivory") y los títulos apropiados.

```
hist(x = banco$duration, main = "Histograma de Duration",
     xlab = "Duration", ylab = "Frecuencia",
     col = "ivory")
```



Como es usual, puedes consultar los demás argumentos de esta función llamando `?hist()`.

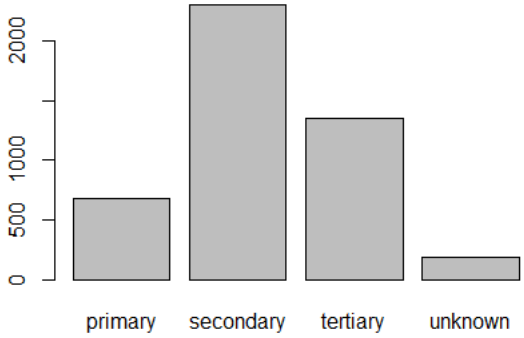
12.4 Gráficas de barras

Este es quizás el tipo de gráfico mejor conocido de todos. Una gráfica de este tipo nos muestra la frecuencia con la que se han observado los datos de una variable discreta, con una barra para cada categoría de esta variable.

La función `plot()` puede generar gráficos de barra si damos como argumento `x` un vector de factor o cadena de texto, sin dar un argumento `y`.

Por ejemplo, creamos una gráfica de barras de la variable educación ("education") de `banco`

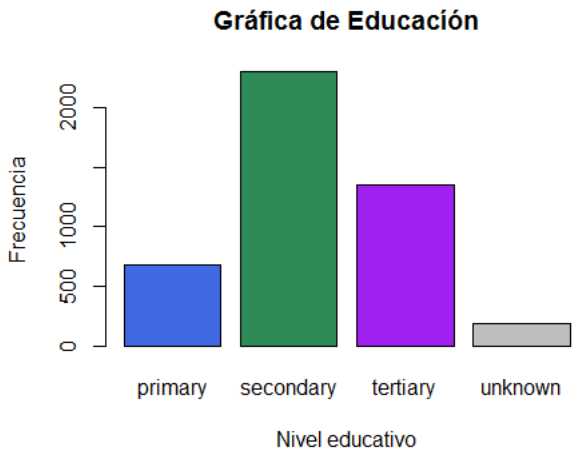
```
plot(x = banco$education)
```



Al igual que con los histogramas, obtenemos un resultado aceptable no obstante el esfuerzo mínimo que hemos hecho para generar nuestra gráfica de barras.

Podemos ajustar los parámetros gráficos con los argumentos `main`, `xlab`, `ylab` y `col`. En este caso, podemos darle a `col` un vector de colores, uno por barra, para que cada una sea distinta.

```
plot(x = banco$education, main = "Gráfica de Educación",
     xlab = "Nivel educativo", ylab = "Frecuencia",
     col = c("royalblue", "seagreen", "purple", "grey"))
```



La combinación de colores puede mejorar, pero ya tenemos una gráfica de barras presentable.

Sin embargo, hay ocasiones en las que deseamos usar gráficas de barras para presentar proporciones, que deseamos barras apiladas. Para esos casos, usamos la función `barplot()`.

12.4.1 La función `barplot()`

Además de usar `plot()`, podemos crear gráficas de barra con la función `barplot()`.

`barplot` pide como argumento una matriz, que represente una **tabla de contingencia** con los datos a graficar. Este tipo de tablas pueden ser generadas con la función `table()`.

`table()` pide como argumento uno o más vectores, de preferencia variables discretas. Si damos sólo un vector como argumento, devuelve un conteo, si damos dos o más variables, devuelve tablas de contingencia.

Por ejemplo, el conteo de la variable **education**,

```
table(banco$education)
```

```
##
##   primary secondary  tertiary   unknown
##       678      2306      1350       187
```

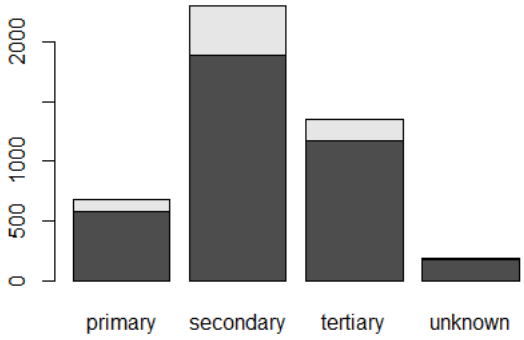
Si damos como argumentos la variable **education** y la variable **loan** (préstamo), obtenemos una tabla de contingencia, que asignaremos al objeto `tab_banco`.

```
tab_banco <- table(banco$loan, banco$education)

# Resultado
tab_banco
```

```
##
##       primary secondary tertiary unknown
##   no      584      1890     1176     180
##   yes      94       416      174       7
```

Damos como argumento `tab_banco` a `barplot()` y nos devuelve una gráfica de barras apiladas.

```
barplot(tab_banco)
```



Si deseamos graficar proporciones en lugar de conteos, usamos la función `prop.table()`.

Esta función nos pide como argumento una tabla de contingencia generada por `table()`. y un número para `margin`. El argumento `margin` es similar a `MARGIN` de `apply()` (como vimos en el capítulo 10).

- Si damos como argumento 1, las proporciones se calcularán agrupadas por renglón. La suma de proporciones por renglón será igual a 1.
- Si damos como argumento 2, las proporciones se calcularán agrupadas por columna. La suma de proporciones por columna será igual a 1
- Si no damos ningún argumento, las proporciones se calcularán usando toda la tabla como grupo. La suma de proporciones de todas las celdas en la tabla será igual a 1.

Para ilustrar esto, veamos los tres casos para `margin` usando como argumento nuestro objeto `tab_banco`.

```
# Proporción por renglón
prop.table(tab_banco, margin = 1)
```

```
##
##          primary  secondary   tertiary    unknown
##   no  0.15248042 0.49347258 0.30704961 0.04699739
##   yes 0.13603473 0.60202605 0.25180897 0.01013025
```

```
# Porporción por columna
prop.table(tab_banco, margin = 2)
```

```
##
##          primary  secondary   tertiary    unknown
##   no  0.86135693 0.81960104 0.87111111 0.96256684
##   yes 0.13864307 0.18039896 0.12888889 0.03743316
```

```
# Porporción por tabla
prop.table(tab_banco)
```

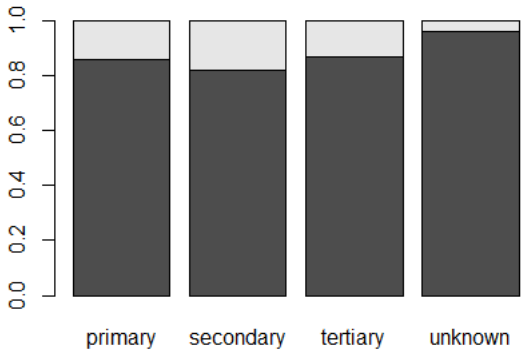
```
##
##          primary  secondary   tertiary    unknown
##   no  0.12917496 0.41804910 0.26011944 0.03981420
##   yes 0.02079186 0.09201504 0.03848706 0.00154833
```

Nosotros queremos obtener las proporciones por columna, así que usaremos `margin = 2`.

```r
ptab_banco <- prop.table(tab_banco, margin = 2)
```

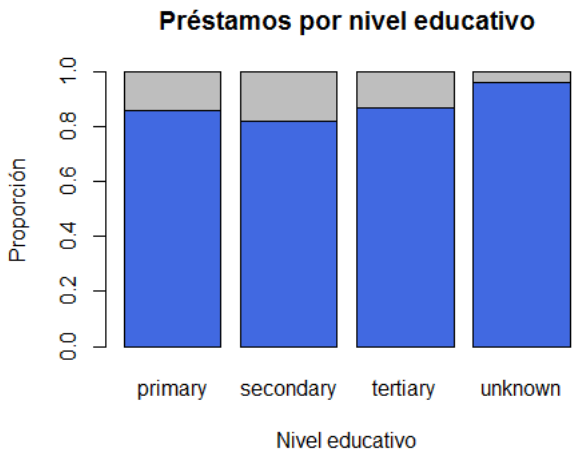
Damos el resultado de la operación anterior a `barplot()`.

```r
barplot(ptab_banco)
```



Hemos obtenido el resultado esperado, pero podemos mejorar la presentación. Nota que con barras apiladas el argumento `col` se puede usar para colorear las categorias al interior de las barras.

```r
barplot(ptab_banco,  main = "Préstamos por nivel educativo",
     xlab = "Nivel educativo", ylab = "Proporción",
     col = c("royalblue", "grey"))
```



Luce bien, pero tenemos un problema: no sabemos qué representan las categorías en nuestras barras apiladas viendo sólamente nuestra gráfica.

Nosotros podemos consultar directamente con los datos, pero una persona que vea por primera vez esta gráfica no tendrá esa opción, reduciendo con ello su utiidad.

Para solucionar este problema, usamos leyendas.

12.5 Leyendas

Las leyendas son usadas para identificar con mayor claridad los distintos elementos en un gráfico, tales como colores y formas.

En R usamos la función `legend()` para generar leyendas. Esta función debe ser llamada después de crear un gráfico. En cierto modo es una anotación a un gráfico ya existente. `legend()` es una función relativamente compleja, así que sólo revisaremos lo esencial.

`legend()` siempre nos pide siempre los siguientes argumentos.

- `legend`: Las etiquetas de los datos que queremos describir con la leyenda. Por ejemplo, si tenemos cuatro categorías a describir, proporcionamos un vector de cuatro cadenas de texto.
- `fill`: Los colores que acompañan a las etiquetas definidas con `legend`. Estos colores tienen que coincidir con los que hemos usado en el gráfico.
- `x` y `y`: Las coordenadas en pixeles, en las que estará ubicada la leyenda. Podemos dar como argumento a `x` alguno de los siguientes, para ubicar automáticamente la leyenda: "bottomright", "bottom", "bottomleft", "left", "topleft", "top", "topright", "right", "center".
- `title`: Para poner título a la leyenda.

Además, tenemos muchos otros argumentos opcionales, que puedes consultar en la documentación llamando `?legend()`.

Vamos a agregar una leyenda a la última gráfica de barras que creamos en la sección anterior de este capítulo.

Entonces necesitamos conocer las etiquetas que daremos como argumento `legend` y a qué colores corresponden al vector `banco$loan`.

Usamos la función `unique` para determinar cuántos valores únicos hay en este vector. Cada uno de estos valores corresponde a una etiqueta. Esta función, si la aplicamos a un vector de tipo factor, nos devuelve sus niveles.

```
unique(banco$loan)
```

```
## [1] no  yes
## Levels: no yes
```

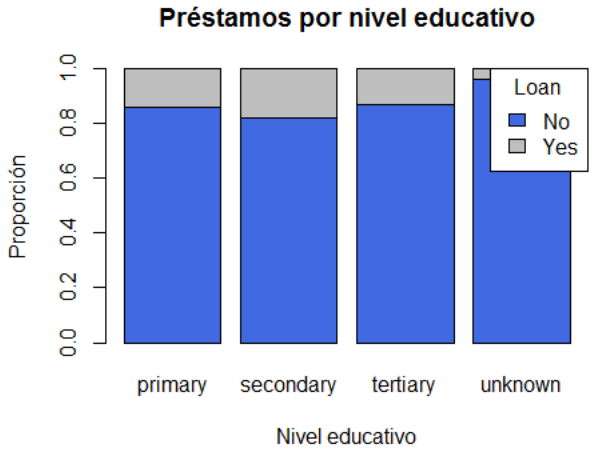
Tenemos dos etiquetas, "no" y "yes" (no y sí, respectivamente), en ese orden, por lo que ese será nuestro argumento `legend`.

Nosotros determinamos los colores en la sección anterior como "royalblue" y "grey", en ese orden. Por lo tanto, tendremos que "no" será coloreado con "royalblue", y "yes" con "grey". como vamos a rellenar una barra, esto colores los daremos al argumento `fill`.

Por último, daremos como "topright" como argumento `x` para que nuestra leyenda se unique en la parte superior derecha de nuestro gráfico.

Aplicamos todo, incluído generar el gráfico al que agregaremos la leyenda.

```
barplot(ptab_banco,  main = "Préstamos por nivel educativo",
     xlab = "Nivel educativo", ylab = "Proporción",
     col = c("royalblue", "grey"))
legend(x = "topright", legend = c("No", "Yes"), fill = c("royalblue", "grey"),
       title = "Loan")
```



Se ve mucho más clara la información, pues ahora estamos mostrando a qué categoría corresponden los colores que hemos empleado en el gráfico.

En las secciones siguientes agregaremos leyendas a otros gráficos, con lo cual quedará un poco más claro el uso de `legend()`.

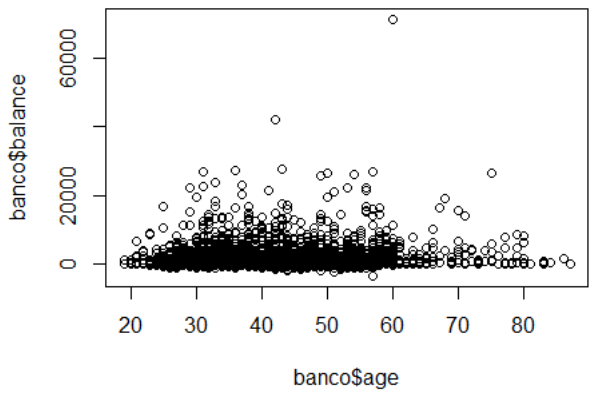
12.6 Diagramas de dispersión

Este tipo de gráfico es usado para mostrar la relación entre dos variables numéricas continuas, usando puntos. Cada punto representa la intersección entre los valores de ambas variables.

Para generar un diagrama de dispersión, damos vectores numéricos como argumentos `x` y `y` a la función `plot()`.

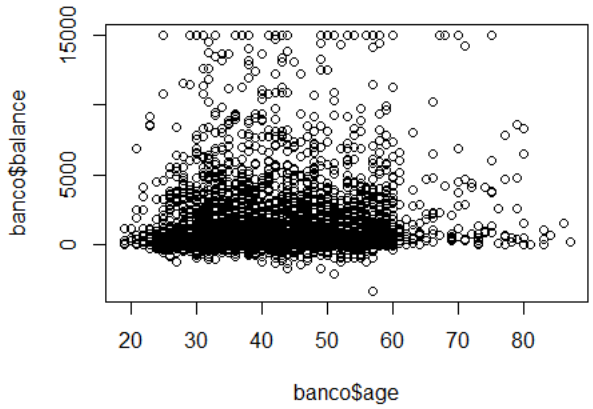
Veamos la relación entre las variables **age** y **balance** de `banco`.

```
plot(x = banco$age, y = banco$balance)
```



Tenemos algunos datos extremos tanto en **balance**. Para fines de tener una gráfica más informativa, vamos a recodificarlos usando `ifelse()`, cambiando todos los valores mayores a 15 000.

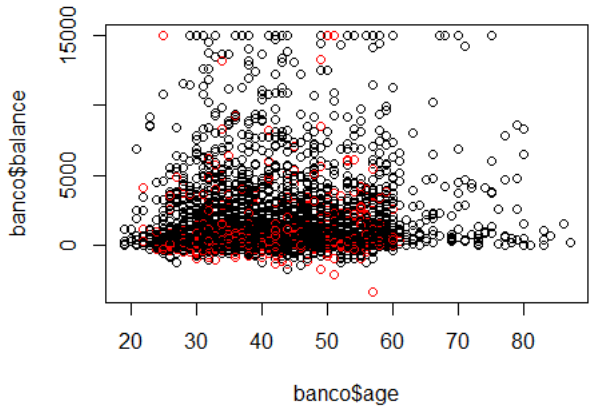
```r
banco$balance <- ifelse(banco$balance > 15000, 15000, banco$balance)

plot(x = banco$age, y = banco$balance)
```



En los diagramas de dispersión, podemos usar el argumento `col` para camiar el color de los puntos usando como referencia una tercera variable.

La variable que usaremos será, de nuevo, **loan**

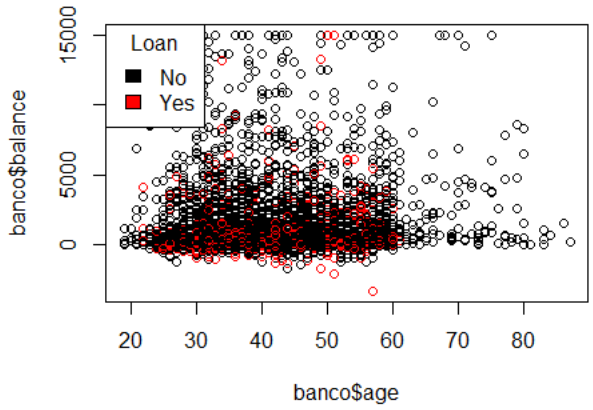
```r
plot(x = banco$age, y = banco$balance, col= banco$loan)
```



Nos sería de utilidad una leyenda para interpretar más fácilmente los colores.

Ya sabemos que los niveles de **loan** son "no" y "yes", además de que los colores han sido rojo y nregro, así que agregar una leyenda será relativamente fácil.

```r
plot(x = banco$age, y = banco$balance, col= banco$loan)
legend(x = "topleft", legend = c("No", "Yes"), fill = c("Black", "Red"), title = "Loan")
```

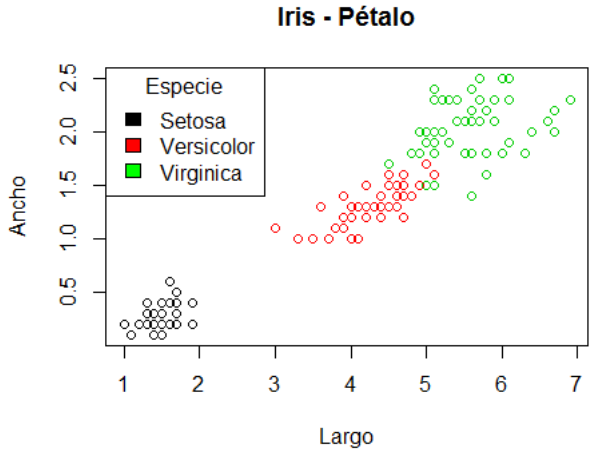


Desafortunadamente esta gráfica no es muy informativa para nuestros datos. Por fortuna, podemos probar con un conjunto de datos diferente.

Si usamos diagramas de dispersión con `iris` obtendremos gráficos mucho más interesantes.

Creamos un gráfico con las medidas de pétalo, aplicando lo que hemos visto para generar diagramas de dispersión.

```
plot(x = iris$Petal.Length, y = iris$Petal.Width, col = iris$Species,
     main = "Iris - Pétalo", xlab = "Largo", ylab = "Ancho")
legend(x = "topleft", legend = c("Setosa", "Versicolor", "Virginica"),
       fill = c("black", "red", "green"), title = "Especie")
```



12.7 Diagramas de caja

Los diagrams de caja, también conocidos como de caja y bigotes son gráficos que muestra la distribución de una variable usando cuartiles, de modo que de manera visual podemos inferir algunas cosas sobre su dispersión, ubicación y simetría.

Una gráfica de este tipo dibuja un rectángulo cruzado por una línea recta horizontal. Esta linea recta representa la mediana, el segundo cuartil, su base representa el pimer cuartil y su parte superior el tercer cuartil. Al rango entre

el primer y tercer cuartil se le conoce como intercuartílico (RIC). Esta es la caja.

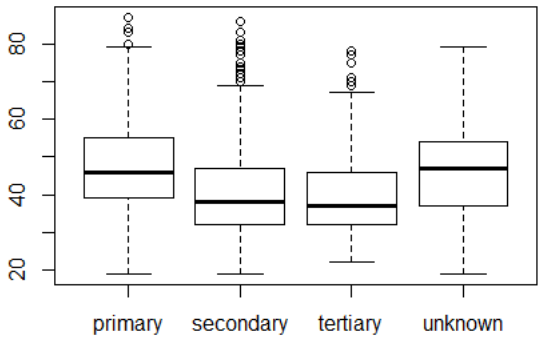
Además, de la caja salen dos líneas. Una que llega hasta el mínimo valor de los datos en la variable o hasta el primer cuartil menos hast 1.5 veces el RIC; y otra que llegar hasta el valor máximo de los datos o el tercer cuartil más hasta 1.5 veces el RIC. Estos son los bigotes.

Usamos la función `plot()` para crear este tipo de gráfico, dando como argumento `x` un vector de factor o cadena de texto, y como argumento `y` un vector numérico.

Una ventaja de este tipo de gráfico es que podemos comparar las distribución de una misma variable para diferentes grupos.

Vamos a ver cómo se distribuye la edad por nivel de educación en nuestro objeto `banco`, esto es, las variables **education** y **age**.

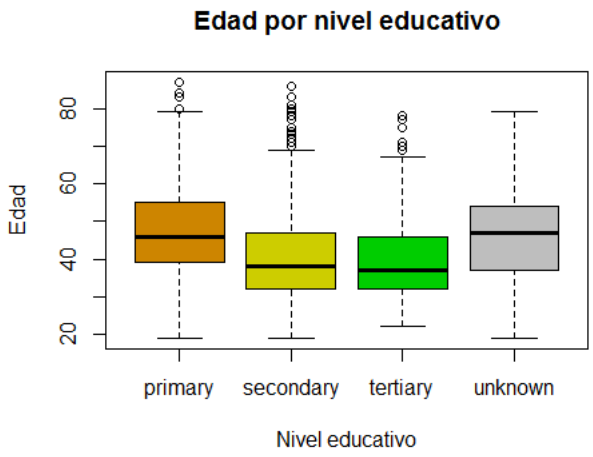
```
plot(x = banco$education, y = banco$age)
```



Podemos ver que las personas con menor nivel educativo tienden a tener una edad mayor. La mayoría de las personas con educación primaria tienen entre 40 y 50 años, mientras que la mayoría con educación terciaria tiene entre 35 y 45 años, aproximadamente.

Por supuesto, podemos cambiar los parámetros gráficos a un diagrama de caja.

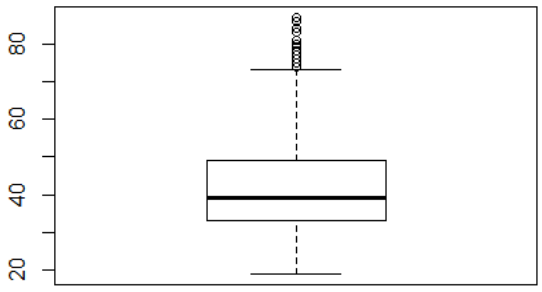
```
plot(x = banco$education, y = banco$age, main = "Edad por nivel educativo",
     xlab = "Nivel educativo", ylab = "Edad",
     col = c("orange3", "yellow3", "green3", "grey"))
```



También podemos crear diagramas de caja con la función `boxplot()`. Esta función puede generar diagramas de caja de dos maneras distintas.

En la primera manera, si damos como argumento `x` un vector numérico, nos dará un diagrama de caja de esa variable.

```
boxplot(x = banco$age)
```

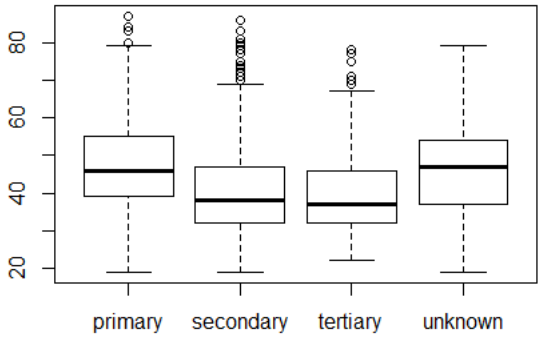


En la segunda manera necesitamos dar dos argumentos:

- `formula`: Para esta función las fórmulas tienen el formato `y ~ x`, donde x es el nombre de la variable continua a graficar, y la x es la variable que usaremos como agrupación.
- `data`: Es el data frame del que serántomadas las variables.

Por ejemplo, para mostrar diagramas de caja por nivel educativo, nuestra variable y es **age** y nuestra variable x es **education**, por lo tanto, `formula` será `age ~ education`.

```
boxplot(formula = age ~ education, data =  banco)
```



12.8 Gráficos de mosaico

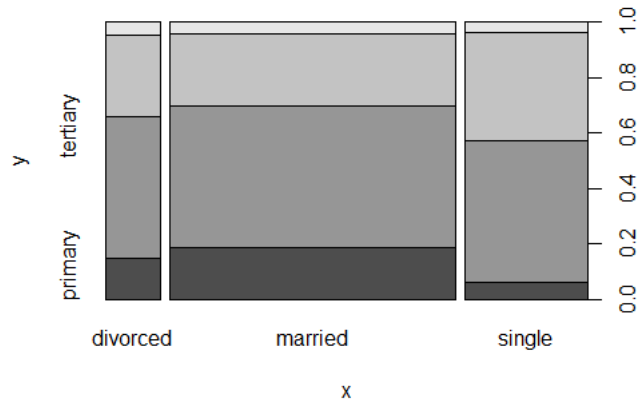
Los gráficos de mosaico o diagramas de Marimekko son usados para mostrar la relación entre dos variables discretas, ya sean factores o cadenas de texto.

Este tipo de grafico recibe su nombre porque consiste en una cuadricula, en la que cada rectángulo representa el numero de casos que corresponden a un cruce específico de variables. Entre más casos se encuentren en ese cruce, más grande será el rectángulo.

Para obtener un gráfico de mosaico, damos como vectores de factor o cadena de texto como argumentos `x` y `y` a la función `plot()`.

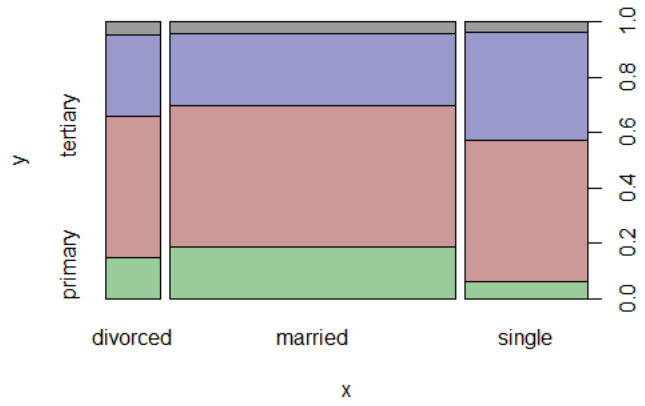
Por ejemplo, intentemos graficar el estado marital con el nivel educativo de las personas en `banco`

```
plot(x = banco$marital, y = banco$education)
```



Podemos cambiar el color de los mosaicos con el argumento `col`. Debemos proporcionar un color por cada nivel del vector en el eje Y.

```r
plot(x = banco$marital, y = banco$education,
     col = c("#99cc99", "#cc9999", "#9999cc", "#9c9c9c"))
```



De esta manera es más claro que el grupo más numeroso de personas son las casadas con educación secundaria y el más pequeño, divorciadas con educación primaria.

12.9 Exportar gráficos

Exportar los gráficos que hemos creado es un proceso que puede parecer un poco confuso.

Cuando llamamos una de estas funciones, le estamos indicando a R que "mande" nuestro gráfico a un **dispositivo gráfico** (*graphic device*) en nuestra computadora, donde podemos verlo, que por defecto es **una ventana en nuestro escritorio** o el panel **Plot** si estás usando *RStudio*.

Una consecuencia de esto es que si creas y lo mandas a un dispositivo gráfico en uso, el gráfico nuevo reemplazará al anterior. Por ejemplo, si usas `plot()` para crear un gráfico, se mostrará en una ventana de tu escritorio, pero si usas `plot()` de generar un gráfico distinto, el contenido de esta ventana será reemplazada con este nuevo gráfico. Lo mismo pasa con todos los dispositivos gráficos.

Además, los gráficos no pueden ser guardados en un objetos para después ser exportados. Es necesario mandar nuestros gráficos a un dispositivo como JPG, PNG o algún otro tipo de archivo que pueda ser almacenado en nuestro disco duro.

Para exportar un gráfico usamos alguna de las siguientes funciones, cada una corresponde con un tipo de archivo distinto. No son las únicas, pero son las más usadas.

- `bpm()`
- `jpeg()`
- `pdf()`
- `png()`
- `tiff()`

Cada una de estas funciones tiene los siguientes argumentos tres argumentos principales.

- `filename`: El nombre y ruta del archivo de imagen a crear. Si no especificamos una ruta completa, entonces el el archivo será creado en nuestro directorio de trabajo.
- `width`: El **ancho** del archivo de imagen a crear, por defecto en pixeles.
- `height`: El **alto** del archivo de imagen a crear, por defecto en pixeles.

La manera de utilizar estas funciones llamándolas **antes** de llamar a una función que genere una gráfica. Al hacer esto, le indicamos a R que en lugar de mandar nuestro gráfico a una ventana del escritorio, lo mande a un dispositivo gráfico distinto.

Finalmente, llamamos a la función `dev.off()`, para cerrar el dispositivo gráfico que hemos elegido, de este modo se creará un archivo y podremos crear más gráficos después.

Por ejemplo, para exportar un gráfico con leyenda como un archivo PNG llamamos lo siguiente. Nota que tenemos que dar la misma extensión de archivo que la función que estamos llamando, en este caso **.png**.

```
png(filename = "loan_age.png", width = 800, height = 600)
plot(x = banco$age, y = banco$duration, col = banco$loan,
     main = "Edad y Duración", xlab = "Edad", ylab = "Duración")
legend(x = "top", legend = c("No", "Yes"), fill = c("Black", "Red"),

       title = "Loan")
dev.off()
```

```
## png
##   2
```

Si aparece un mensaje como el siguiente, es que hemos tenido éxito.

null device 1

Podemos ver el resultado usando `file.show()`.

```
file.show("loan_age.png")
```

De esta manera podemos exportar cualquier tipo de gráfico generado con R.