Fundamentos de R

José C. Chacón

Fundamentos de R

José C. Chacón

José C. Chacón
Universidad Complutense de Madrid

Fundamentos de R

PUBLICADO POR EL AUTOR

ISBN: 978-84-09-32811-6

Puede descargarse desde https://leanpub.com/fundamentosder

Las soluciones a los ejercicios del manual pueden encontrarse en *Fundamentos de R: ejercicios resueltos y comentados*, disponible gratuitamente en https://leanpub.com/fundamentosder_ejercicios.

A Belén,

quien ha compartido muchas de las

horas de vida dedicadas a este libro

Índice general

Antes de empezar...

R es una herramienta de análisis y representación de datos de extraordinaria potencia y un lenguaje de programación completo, lo que permite, en el ámbito del análisis de datos, hacer prácticamente cualquier cosa. El precio a pagar es un aprendizaje lento[1] aunque, por contra, la recompensa es doble: por un lado, un control prácticamente absoluto sobre los datos y su tratamiento, análisis, representación, almacenamiento y exportación; por otro, el libre acceso a una cantidad ingente de recursos y herramientas libres, gratuitas y de calidad.

Es importante dejar claro desde ahora que el objetivo del manual es *aprender R*, y no *aprender a hacer análisis estadísticos con R*. Así, de los dos componentes del análisis de datos, las técnicas estadísticas y la herramienta para implementarlas, sólo trataremos el segundo. Hay multitud de libros sobre todo tipo análisis de datos con R,[2] pero no tantos que se centren en la propia herramienta. El resultado, en muchos casos, es que el analista se ve limitado por su conocimiento de R. Y es que cualquier labor profesional requiere de un conocimiento igualmente profesional de las herramientas utilizadas. Así, la calidad de los análisis a realizar, y su eficiencia, dependerán de la profundidad con que conocemos la herramienta y nuestra fluidez al usarla. Lo que nos lleva al siguiente punto.

Hay tres formas de aproximarse a R según nuestra experiencia y la mayoría comienza, de forma natural, por la necesidad de hacer algún análisis estadístico: se obtiene una idea *ligera* de R, lo justo para hacer algún análisis de interés, y vamos luego poco a poco aprendiendo el resto, mientras se siguen haciendo análisis.[3] Por desgracia, en cuanto salimos de los ejemplos básicos empiezan los problemas. No es fácil entender el funcionamiento de R sin una buena organización de los conceptos y con la profundidad que requieren.

El resultado suele ser uno de tres: el abandono («R es muy difícil»); el *retro-aprendizaje* (aprender los conceptos y objetos que *ya* estamos usando),[4] o comenzar, cuando se comprende su necesidad, *un aprendizaje desde cero, completo y coordinado*. Esta última aproximación es la que usaremos aquí.

[1] Decimos «lento», y no con una «curva de aprendizaje empinada» en tanto una curva tal proporcionaría un gran aprendizaje en poco tiempo, como se ve. La segunda curva se ajusta mejor a la realidad.

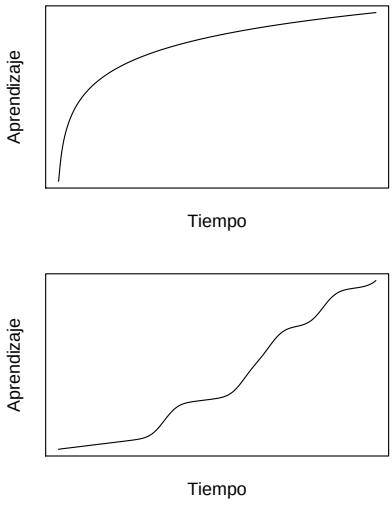

Figura 1: Dos curvas de aprendizaje, la primera de ellas, *empinada*.

[2] Prácticamente todos con una breve introducción al lenguaje R.

[3] Es frecuente cuando nos iniciamos con manuales del estilo de *Análisis de* tal tipo *con R*.

[4] Generalmente para entender qué falla y cómo arreglarlo. Es un aprendizaje lento, desorganizado y con muchas lagunas.

Aprender a programar desde cero, en R o cualquier otro lenguaje, no es fácil. Quien ha programado con anterioridad lo sabe. En caso contrario, es bueno saber que el aprendizaje abarcará dos áreas diferentes aunque siempre entremezcladas.

Por un lado, y como en un lenguaje natural, hay que aprender una semántica (para nombrar los objetos, junto con su significado y uso) y una sintaxis (cómo se componen correctamente las instrucciones en R). Pero a diferencia de los lenguajes naturales, la más mínima alteración de cualquiera de ellas (una letra, una coma, un paréntesis) genera un error. Detectarlo no es siempre fácil y se tarda un tiempo en aceptar que el problema no es de R ni del manual.[5]

[5] Como suele oírse en el ámbito informático, «maldito ordenador, que no hace lo que quiero sino lo que le digo».

Pero hay otro lado, menos conocido. Y es que la parte más difícil (y más interesante) no es escribir código, sino *traducir la solución del problema* a términos del lenguaje de programación utilizado. Ello requiere comprender el problema, determinar sus componentes y las relaciones entre ellos y traducir estos componentes en objetos del lenguaje y las relaciones en funciones que se aplicarán a esos objetos. Para ello habrá que adquirir y/o potenciar un pensamiento analítico, disciplinado y lógico que no puede sino mejorar el mobiliario de cualquier cabeza. Precisamente muchos de los ejercicios inciden en este aspecto, el gran olvidado al practicar un lenguaje. Y es que...

Sin práctica no hay aprendizaje, por ello el manual está plagado de ejercicios. La mayoría se dedica a aplicar lo explicado y en otros casos se extienden más allá, lo que requiere conocer alguna nueva función o hacer una búsqueda en la *web*. En general, se ha intentado explicar las respuestas cuando lo requieren, pero no sólo eso: en algunos casos, se ofrecen variantes para conseguir un mismo propósito, incluso mostrando respuestas erróneas comunes y analizando sus diferencias. El contraste ayuda a entender, y se aprende más forzando una función a dar error que usándola correctamente sin más: sabiendo cómo y por qué falla, sabremos cómo hacer que no falle.

Las soluciones están disponibles en los *Ejercicios resueltos y comentados*,[6] pero es bueno recordar algo que ya sabemos. Dedicar media hora a resolver un ejercicio simple para descubrir que el error era un despiste o *una tontería* no es una pérdida de tiempo. Cuando se observa la solución, y pese a que podamos sentirnos algo idiotas, hemos conseguido mucho: la probabilidad de que repitamos *ese* error es mínima.[7] Por contra, mirar la solución a la primera dificultad sólo mostrará un resultado evidente. No reducirá la autoestima (no ha dado tiempo) pero no habremos aprendido nada. Eso sí es una completa pérdida de tiempo.

[6] Disponible gratuitamente en https://leanpub.com/fundamentosder_ejercicios.

[7] Esto es, hemos *aprendido*.

En cuanto a la organización, podríamos dividir el texto en dos grandes áreas: una primera que trata de los objetos que almacenan información y otra que cubre el procesamiento de tales objetos y la

información contenida en ellos. Si entramos en más detalle, tenemos una parte inicial, *primeros pasos*, que muestra una visión general de R (capítulos 1 y 2) y un primer acercamiento al lenguaje.

Entrando ya en los *fundamentos*, los capítulos 3 y 4 exponen los conceptos mínimos para conocer y controlar nuestro entorno y los vectores.[8] A partir de ahí vamos estudiando los vectores atómicos (capítulos 5 al 10) y no atómicos (11 y 12), y algunos valores especiales (cap. 13), seguidos por los capítulos dedicados a la programación básica: lectura y escritura de archivos (14), estructuras de control (15), funciones (16) y manipulaciones (17). Continuamos con capítulos dedicados a los paquetes (18) y a los gráficos (19), dos de las grandes fortalezas de este lenguaje y acabamos este apartado con otro capítulo dedicado al control (20) y la programación orientada a objetos (21).

La tercera parte, *aplicaciones*, es breve. Un capítulo sobre estadística (capítulo 22) se dedica a entroncar las fases de un análisis estadístico con las herramientas, paquetes y demás recursos disponibles para R. Otro (23) se centra en la probabilidad y el muestreo, mientras que el último (24) ofrece varias vías de entre las muchas posibles para seguir aprendiendo R.

Los *apéndices* complementan todo lo anterior con aspectos prácticos que no siempre reciben la atención que merecen. El apéndice A está pensado para quienes puedan necesitar ayuda en la instalación, actualización o para conocer mejor la interfaz que utilizaremos. El apéndice B ofrece recomendaciones para que nuestro código sea más legible, menos propenso a errores y más fácil de modificar. Por último, el apéndice C completa las recomendaciones anteriores, pero esta vez en relación a la estructura de la información contenida en los archivos de datos.

[8] Uno de los ladrillos fundamentales de R, junto con las funciones.

Forma y contenido. Este manual nació como complemento a actividades docentes[9]. Y lo primero que un docente echa en falta al escribir, además de la natural interacción, son las preguntas[10] y, con ellas, la posibilidad de esos comentarios frecuentes en cualquier clase; desde un simple «esto se ampliará en el próximo capítulo» hasta ejemplos adicionales, enlaces con lo ya visto o comentarios laterales más extensos, incluyendo accesos a otros recursos para ilustrar lo explicado. Su ausencia resulta extraña, y resta tridimensionalidad a las explicaciones y a la *red* de conceptos y relaciones que se dan de forma natural en la docencia.

Una solución parcial (y elegante) es el *formato Tufte*, llamado así por Edward R. Tufte (1997; 2001; 2006; 2013)[11], quien lo ideó con objeto de permitir comentarios e ilustraciones extensos[12].

El resultado final es el que podemos observar: una columna principal que contiene el desarrollo de los temas y un amplio margen con frecuentes comentarios, aclaraciones, enlaces a otros temas o ampliaciones, ilustraciones y ejemplos y, cómo no, cientos de enlaces a la

[9] Para la asignatura *Métodos informáticos*, impartida en la Universidad Complutense de Madrid dentro del Máster Interuniversitario en Metodología de las Ciencias del Comportamiento y de la Salud (`http://www.metodologiaccs.es`).

[10] Que en unas ocasiones hemos añadido retóricamente y, en otras, a través de ejercicios.

[11] Edward R. Tufte es un profesor de ciencia política y estadística y pionero y auténtico especialista en la visualización de información (en campos como *information design* o *visual literacy*). Es conocido por conceptos como la *data-ink ratio* para referirse a la cantidad de información por unidad de tinta (Tufte, 1990), o por resumir sus críticas a *PowerPoint* con la frase "*Power corrupts. PowerPoint corrupts absolutely*" (Tufte, 2003).

[12] Y no sólo por parte del autor; los márgenes dan espacio para comentarios y anotaciones, en papel o electrónicos, de los lectores.

web. Si se dispone de tiempo para ampliar y explorar, estos enlaces permitirán conocer, más allá del lenguaje R, el *mundo* de R.

Modos de lectura. El origen docente de este manual le da una orientación muy específica, aunque hay otros modos de lectura posibles. En total, se diría que tres: Uno, el original, con la lectura y resolución de todos los capítulos y ejercicios, lo que casi con seguridad llevará a una sólida formación en R.

Dos, una lectura más ágil, que puede consistir en los apartados iniciales de los capítulos. Se observará que los capítulos entran en cuestiones más complejas o detalladas conforme avanzan[13], y cada lector puede decidir dónde pasar al capítulo siguiente. El aprendizaje tendrá algunas lagunas y es probable que la práctica haga necesario volver para completar la información.

[13] Incluyendo algunos apartados que comienzan con *Avanzado:...*, lo que indica que el nivel puede superar el esperable en ese momento, pero puede convenir en una lectura posterior.

Tres, como manual de consulta. Si el índice y la organización son adecuados, es fácil acceder a cualquier aspecto de todo lo tratado y localizar ejemplos, características... Y el lector decidirá hasta dónde quiere informarse de tal o cual función, objeto o procedimiento.

También es posible utilizarlo como manual de referencia en un curso de R. En tal caso, la estrategia óptima, al menos en mi experiencia, ha sido la lectura *previa* (con sus ejercicios) de algunos capítulos y posteriormente, en clase, hacer un buen repaso, aclaración, solución de dudas y realización de más ejercicios. Los alumnos saben leer, así que la interacción en clase puede dedicarse a dinámicas más productivas.

Y no olvido los agradecimientos. Los primeros, para las varias hornadas de alumnos del Máster de Metodología de las Ciencias del Comportamiento y de la Salud, que disfrutaron/sufrieron cada versión de este manual en la asignatura de Métodos informáticos. Ellos y ellas indicaron errores, mejoras y me animaron a seguir escribiendo cuando parecía que era una tarea imposible de acabar.

Pero los alumnos son tolerantes, o probablemente no se atrevan a decirme, directamente, cosas del tipo de «mira, este capítulo es un completo desastre; no hay quien se entere de nada».[14] Que es lo que vino a decirme María José Hernández Lloreda, amiga y compañera, de algunos capítulos iniciales. Y que era exactamente lo que necesitaba: otra mirada que me llevara a salir de mi burbuja. Así que si los capítulos iniciales se entienden, algo imprescindible en un manual de este tipo, denle las gracias a ella. Otros compañeros también revisaron el manual, o incluso lo han utilizado en algunas clases, como Miguel Á. Castellanos. Gracias también a todos ellos.

[14] Especialmente si aún tengo que evaluarles.

Pero el mayor agradecimiento es para Belén. Apoyar (y animar) a alguien que escribe un libro en fines de semana, vacaciones y ratos libres durante años es un tostón... por decirlo muy suavemente. Si sabes de qué hablo, sabes cómo es; si no, da igual lo que te cuenten.

Y a pesar de estas ayudas, quedarán erratas e incluso algún error. En ambos casos, y como no podía ser de otra manera, toda la responsabilidad es del autor.

Nada más; esperamos ayudar a aprender `R` e, idealmente, que se disfrute del recorrido. Suerte.

José C. Chacón
Colmenarejo, agosto de 2021

PARTE I: Primeros pasos

1
R: Características, historia y recursos

Comenzaremos este manual dando una definición general de R, pasaremos por algunos datos históricos y llegaremos hasta la instalación y arranque del programa. A continuación se resumen los contenidos de este capítulo.

RESUMEN

Qué es R (1.1). Un programa, o sistema, o entorno... Veremos una idea general de qué tipo de herramienta tenemos ante nosotros.

Algo de historia (1.2). El origen de las frecuentes alusiones a S, y las implicaciones de esta relación.

Recursos (1.3). ¿Dónde está R? ¿Qué hay disponible y dónde y cómo acceder? ¿Cómo resolver mis dudas o buscar materiales?

Instalación (1.4). Cómo instalar R y RStudio.

Comenzar a usar R (1.5). Abrir el programa y qué encontramos.

1.1 Qué es R

Podemos definir R como un entorno, esto es: un sistema coherente y completamente planificado[1]. Como todo sistema, incluye varias partes y funciones, que en el caso de R son:

- un lenguaje de programación completo,
- medios para la lectura, almacenamiento y escritura de datos,
- un conjunto de utilidades para la manipulación, cálculo y análisis de los datos y
- herramientas para la creación de gráficos.

Simultáneamente, R es un *dialecto* del lenguaje S, como se verá posteriormente. Y, al igual que S, R nace y se orienta principalmente al análisis estadístico de datos y su representación gráfica.

[1] Véase en la web del proyecto R https://www.r-project.org/about.html

Desde el punto de vista de la interacción con el usuario, R es y funciona como un lenguaje de programación, algo imposible de olvidar en tanto toda la comunicación con R es a través de instrucciones de ese lenguaje. Así, para quien no ha programado nunca, R es una sorpresa. No hay menús que permitan abrir datos o hacer un determinado análisis[2].

Las secuencias de acciones (abrir archivo de datos, depurarlos, obtener algunos descriptivos y gráficos, hacer análisis, etc.) pueden ser similares a las de otros programas de análisis mediante menús, pero hay al menos cinco diferencias fundamentales:

[2] Existen interfaces variadas que trabajan sobre R de forma más o menos visible. Pueden consultarse en el apartado A.3.3 del primer apéndice.

1. Esta secuencia, como se dijo, toma la forma de un diálogo.
2. Los resultados de cada acción pueden almacenarse como nuevos objetos y reutilizarse.
3. El grado de control y detalle en las manipulaciones, acciones y especificaciones es generalmente mucho mayor.
4. Se pueden programar análisis o cálculos no disponibles.
5. Existen miles de paquetes adicionales, con infinidad de funciones disponibles para todas las tareas que uno pueda imaginar.

R es un proyecto GNU, esto es, uno de los muchos que participan en el proyecto de colaboración en software libre[3], fundado en 1983 por Richard M. Stallman[4] en el MIT. Ello que implica que está disponible gratuitamente bajo la *GNU General Public License* para varios sistemas operativos. El código fuente para el entorno de software R está escrito principalmente en C y R, y también en Fortran.

[3] https://en.wikipedia.org/wiki/GNU_Project

[4] https://en.wikipedia.org/wiki/Richard_Stallman

R utiliza programación funcional y orientada a objetos. La programación funcional[5] se basa en las llamadas a funciones, que encapsulan procedimientos habituales, y permiten construir programas complejos a partir de módulos sencillos y robustos.

La programación orientada a objetos[6] permite, a través de las llamadas *funciones genéricas*, modificar el comportamiento de una misma función según la clase del objeto al que se aplica.

[5] Véase, por ejemplo, https://es.wikipedia.org/wiki/Programaci%C3%B3n_funcional

[6] Para más detalles, vease https://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos.
En realidad, R tiene tres sistemas de programación orientada a objetos, según se trate de objetos de tipo S3, S4 o RC. En el presente manual sólo trataremos en detalle los objetos de tipo S3.

Por ahora, y para comenzar, basta añadir que el lenguaje de R es interpretado, lo que quiere decir que, para poder ejecutarse, debemos tener abierto el entorno de R[7]. Ello implica, como ya se dijo, que escribiremos instrucciones y R responderá con alguna acción. Estas instrucciones pueden ser escritas y ejecutadas una a una, o pueden almacenarse como una secuencia ordenada en un archivo y ejecutarse todas de una vez. En tal caso hablamos de un programa, y en R se nombran de la forma `programa.R`.

[7] Los programas en R no pueden, por tanto, compilarse y generar archivos ejecutables independientes del entorno R. Sobre la diferencia entre unos y otros lenguajes se hablará, brevemente, en el apartado 20.6 sobre eficiencia.

1.2 Para entender el presente, algo de historia

A veces sorprende que en la ayuda, en referencias o al hablar de manuales básicos se mencione una y otra vez el lenguaje S. La pregunta evidente es ¿estamos trabajando con R o con S? La respuesta, como ya adelantamos, es que R es un *dialecto* del lenguaje S.

El primer paso en la historia de R es, por tanto, el lenguaje S[8], un programa/entorno desarrollado inicialmente por John Chambers, Rick Becker y Allan Wilks allá por 1975-1976 en los Laboratorios Bell[9]. S constituyó una novedad en su enfoque del análisis de datos, basándose en procesos modulares donde la salida de un módulo alimentaba al siguiente, siguiendo la secuencia natural de los análisis. En 1998, la *Association for Computing Machinery* (ACM) premió a John M. Chambers con el *Software System Award*[10] por la contribución que S supuso dentro del mundo de la estadística aplicada.

Actualmente existen dos implementaciones (o dialectos) del entorno S original: R, parte del proyecto de software libre GNU, y S-PLUS, un producto comercial de TIBCO Software.

[8] https://en.wikipedia.org/wiki/S_(programming_language)

[9] Puede verse un breve resumen de la historia de S en http://ect.bell-labs.com/sl/S/history.html.

[10] http://www.webcitation.org/6iGQuou4m

De hecho, la *ACM* consideró que S «*. . . will forever alter the way people analyze, visualize, and manipulate data [. . .] S is an elegant, widely accepted, and enduring software system, with conceptual integrity, thanks to the insight, taste, and effort of John Chambers.*»

R se inicia cuando, en 1993, Ross Ihaka y Robert Gentleman decidieron crear, para sus clases, un entorno estadístico, *simplemente un banco de pruebas* que implementara los métodos de Scheme Lisp y una sintaxis similar a la de S. En una descripción detallada del proceso[11], Ihaka explica cómo en 1995 dieron acceso ftp al código fuente bajo los términos de la *Free Software Foundation's GNU general license*. En 1996 se inició una lista de correos para *r-testers* a la que siguieron otras (*r-announce*, *r-help* y *r-devel*), pero el crecimiento tanto del propio programa como de los usuarios y sus informes les obligó a crear en 1997 una estructura de almacenamiento y un grupo que gestionara el proyecto completo, dando lugar a un equipo principal (actualmente el *R Core Team*[12]) responsable de los cambios que se realizan en R.

[11] https://www.stat.auckland.ac.nz/~ihaka/downloads/Interface98.pdf

[12] https://www.r-project.org/contributors.html

1.3 Recursos

La fuente principal en todo lo referente a R es la web del proyecto R[13], donde hay información general y acceso a recursos variados, y la CRAN (*The Comprehensive R Archive Network*, o Red completa de archivos de R y sus *mirrors*), más orientada al almacenamiento y distribución del *software* de R. Es recomendable visitarlas y darse un paseo por los distintos apartados.

[13] http://www.r-project.org/

Respecto al software necesario, hay tres elementos principales:

- El propio software de R, la llamada *distribución base*, que puede obtenerse de cualquier *Mirror* de la CRAN[14], de entre los distri-

[14] http://cran.r-project.org/

buidos por todo el mundo. Deberemos elegir la descarga para nuestro sistema operativo (disponible en la actualidad para Linux, macOS[15] y Windows).

[15] Anteriormente MacOS X.

- También son herramientas fundamentales los paquetes, o extensiones. Puede accederse a ellos a través del enlace *Package*[16] en la página principal de la CRAN. A fecha de hoy (15/08/21), el repositorio CRAN contabiliza 17974 paquetes disponibles.

[16] https://cran.r-project.org/web/packages/

- `RStudio` es un IDE (*Integrated Development Environment*, o Entorno de desarrollo integrado) que puede descargarse desde su web[17]. Como su nombre indica, se trata de un entorno (una interfaz, pero también mucho más) que facilita enormemente la gestión de toda la información a la hora de trabajar con `R`.

[17] http://www.rstudio.com/

Respecto a la documentación existen diferentes fuentes. La principal es la misma web del proyecto `R`, donde podemos descargar todo lo necesario para aprender y trabajar con `R`. En el apartado *Documentation*, a la izquierda de la página principal encontramos el enlace *Manuals*[18], donde disponemos de gran número de manuales. Los dos primeros de la siguiente lista son probablemente los más básicos y utilizados (Venables et al., 2019; Paradis, 2002), e incluimos también una tarjeta resumen (Short, 2004)[19]:

[18] http://cran.r-project.org/manuals.html

[19] Los dos primeros manuales, y otros muchos, están disponibles en español en el apartado *Contributed* de la CRAN: https://cran.r-project.org/other-docs.html

- Venables, W. N., Smith, D. M. & The R Core Team (2012). *An introduction to R*. Disponible en http://www.math.vu.nl/sto/onderwijs/statlearn/R-Binder.pdf.
- Paradis, E. (2005). *R for Beginners*. Montpellier: University of Montpellier. Disponible en http://cran.r-project.org/doc/contrib/rdebuts_en.pdf.
- Short, T. (2004). *R Reference Card*. Disponible en https://cran.r-project.org/doc/contrib/Short-refcard.pdf.

La web es otra fuente inagotable de recursos. Frente a una lista demasiado larga, preferimos mostrar sólo dos referencias:

- *Quick R*[20] es una *web* que ofrece una guía rápida, de gran organización y eficiencia. Para cualquier concepto básico ofrece una explicación clara, uno o varios ejemplos de uso y, con frecuencia, enlaces a información más avanzada o específica.
- Pero si tenemos cualquier duda, por simple o compleja que sea, podemos encontrar la solución escribiendo en un buscador "*R how to...*" e indicando el problema o el resultado deseado. En un porcentaje muy alto de los casos, la respuesta estará en *Stack Overflow*[21], que se ha convertido en el lugar de referencia para soluciones, explicaciones y recursos varios en `R`.

[20] http://www.statmethods.net/

[21] https://stackoverflow.com/questions/tagged/r

1.4 Instalar R y RStudio

La instalación básica de R es bien simple, en tanto se trata únicamente de ejecutar el archivo descargado de la CRAN. Salvo casos especiales, es recomendable usar el directorio por defecto.

La instalación de RStudio es igualmente simple: basta ejecutar el archivo descargado, que localizará la instalación de R disponible.

Si la breve información anterior no ha sido suficiente para instalar los programas indicados puede ser conveniente acudir al apartado A.1 del Apendice A, que detalla la descarga e instalación de ambos.

1.5 Comenzar a usar R

Tras la instalación podemos acceder a R de dos formas:

- Pulsando en el icono de R, a la izquierda en la figura 1.1, lo que abre la ventana principal (la *consola*) de R.
- A través de RStudio, haciendo click en el icono a la derecha de la figura 1.1, lo que abre el IDE que usaremos a lo largo de todo el manual.

Figura 1.1: Iconos para R y RStudio. En este manual usaremos RStudio en todo momento.

La principal ventaja de una interfaz *integrada* es precisamente eso: que las diferentes ventanas están presentes y organizadas[22]. Una vez abierta, la interfaz de RStudio muestra una apariencia similar a la de la figura 1.2.

[22] Y hay otras: funciones adicionales que facilitan múltiples tareas, como las utilidades para gestionar este manual.

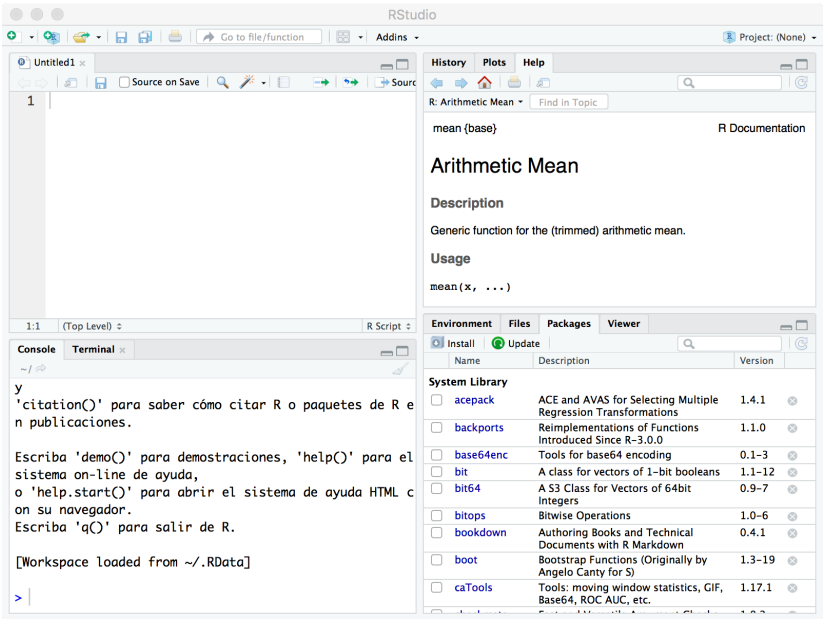

Figura 1.2: Ventana de RStudio con los cuatro paneles.

Aunque las pestañas y su distribución pueden variar[23], el entorno de RStudio siempre contendrá varias pestañas básicas de uso común. Entre ellas observamos la ventana de programa (*source panel*), arriba a la izquierda, y debajo aparece la consola. Esta última servirá para

[23] De hecho, podemos configurar qué ventanas queremos mostrar y en qué panel estará situada cada una en el menú `Tools/Global Options/Pane Layout`.

introducir y ejecutar instrucciones una a una, mientras que la primera permite editar y almacenar un conjunto de intrucciones, lo que solemos llamar un *programa*.

A la derecha se muestran otras dos ventanas habituales: arriba, la ayuda, mostrando información sobre la función `mean()`; debajo, la pestaña *Packages*, que muestra los paquetes instalados.

En el capítulo siguiente haremos un recorrido para ver `R` en funcionamiento y conocer algunas de sus características. Para introducir el código deberemos usar la ventana de consola, que en la figura 1.2 aparece abajo a la izquierda.

No obstante, puede ser recomendable visitar el apartado A.3.2 en el apéndice A. En él se comentan algunas características de `RStudio` que son de utilidad desde el primer día. O también podemos consultarlo en otro momento, para conocer mejor la interfaz.

Ejercicio 1.1.

Localiza y recorre la web del proyecto `R` si aún no lo has hecho. Explora sus apartados, sus paquetes, hojea su revista (*The `R` Journal*)...

Ejercicio 1.2.

Busca información sobre John M. Chambers, Ross Ihaka y Robert Gentleman, o recorre los enlaces que aparecen a lo largo del capítulo.

Ejercicio 1.3.

Localiza y explora la web de la CRAN y ve al apartado de descarga para tu sistema operativo. Localiza la versión adecuada y, si no lo has hecho aún, descárgala e instálala. Comprueba el número actual de paquetes disponibles. Pulsa sobre alguno que te parezca de interés y explora la información que ofrece.

2
Una primera inmersión

En este capítulo introduciremos los principales elementos que se ponen en juego al trabajar con R y que serán tratados a lo largo de todo el manual. Siendo introductorio, no podremos entrar en detalle en ninguna de las cuestiones tratadas; el objetivo es ofrecer una idea de conjunto, amplia y poco definida. Sí que merecen atención especial los objetos tratados (vectores y funciones) y sus características, ya que es sobre ellos, y su uso, de lo que hablaremos en estas páginas.

RESUMEN

Acceso a R y primeras tareas básicas (2.1). Elección del modo de trabajo; los primeros cálculos, expresiones y asignaciones.

Dos conjuntos de objetos *fundamentales en R.* En general, las estructuras de datos, o vectores (2.2), almacenan la información y las funciones (2.3) las procesan.

Archivos, tipos, creación y acceso (2.4). Tipos de archivos, junto con su lectura y escritura, importación y exportación.

Paquetes (2.5). Dada su importancia, un primer acercamiento práctico.

La ayuda (2.6). Imprescindible, fuentes de información internas y externas.

2.1 *Acceso a R y primeras tareas*

El capítulo anterior aludió a que el trabajo con R es *interactivo*, esto es, que toda acción se realiza mediante comandos o instrucciones. Pues bien, una vez abrimos R a través de la interfaz de RStudio, existen dos lugares desde los que puede realizarse ese trabajo interactivo, con usos muy diferentes.

La consola. Principalmente para pruebas o para obtener información específica, momentánea.

Un archivo. Para una secuencia de acciones organizada y preferible siempre que se quiera volver sobre el código ejecutado.

Al hablar de una secuencia de acciones organizadas podemos referirnos a los múltiples pasos de un análisis estadístico, pero puede tratarse también de los ejemplos de este manual, que podemos copiar y pegar en un archivo y guardarlo con un nombre, por ejemplo `Cap_02.R`. En la figura 2.1 vemos, a la izquierda, el archivo `Cap_02.R`, donde aparecen los primeros ejemplos de este capítulo[1]. A la derecha tenemos los mismos ejemplos, ejecutados en la consola. Es claro que el archivo permite guardar la información y revisarla o reutilizarla posteriormente mientras que lo ejecutado en la consola se pierde[2].

[1] Véase que el carácter `'#'` se utiliza para incluir comentarios, títulos, aclaraciones... que no se ejecutan.

[2] En realidad queda registrado en el *historial* (véase la pestaña *History*).

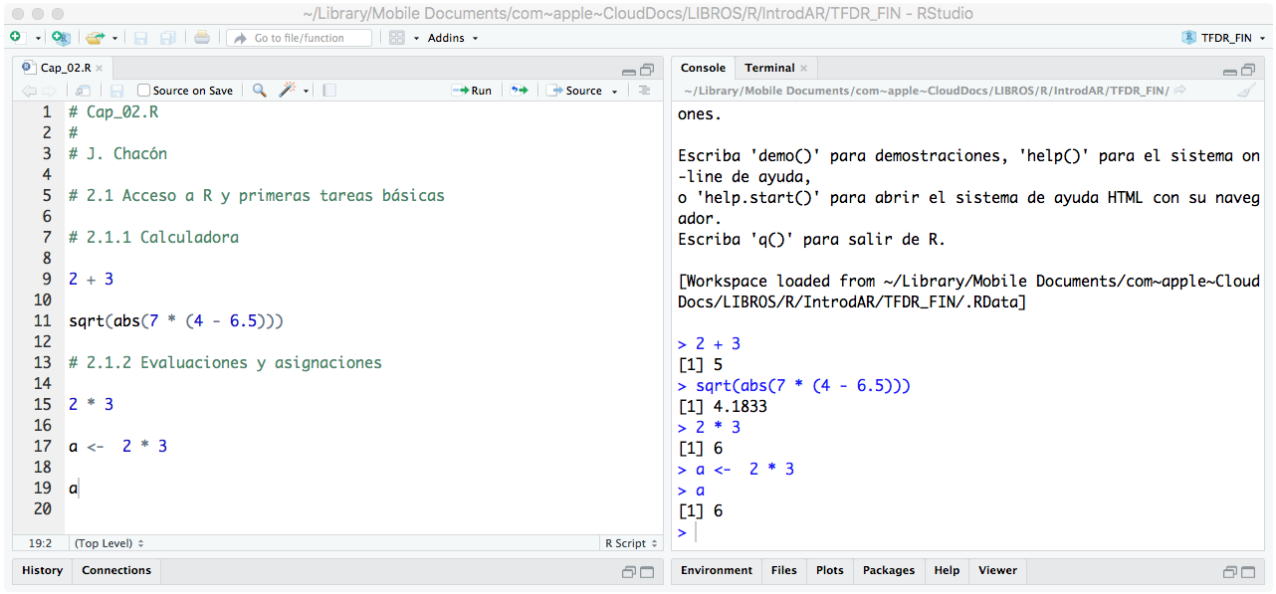

Figura 2.1: Ventanas de archivo (izquierda) y consola (derecha) en el editor de `RStudio`. .

Si usamos la consola, debemos escribir el código deseado (o bien copiarlo y pegarlo) y pulsar *Enter* para ejecutarlo. Si trabajamos con un archivo, pulsar *Enter* sólo servirá para introducir un salto de línea; si queremos ejecutar una línea de código basta con situarnos en algún lugar de ella y pulsar *Control* + *Enter*. Esta misma combinación permite ejecutar un bloque de código previamente seleccionado.

En la consola de `R`, las instrucciones aparecen precedidas del *prompt* (habitualmente el carácter '>'), mientras que las salidas aparecen debajo. En este texto se ha eliminado el *prompt* de las instrucciones (al igual que en un archivo, con objeto facilitar la tarea de copiar y pegar), que aparecen siempre sobre un fondo de color[3]. Las salidas, por su parte, vienen precedidas de los símbolos "##" para identificarlas claramente.

[3] O gris claro en la versión impresa.

Pensemos que *no existe mejor manera* de aprender el lenguaje `R`[4] que *usarlo*: como en el ejemplo, podemos crear un archivo `.R` y pegar en él el código copiado de los ejemplos del manual. Después se recomienda *modificarlo*. Hay que explorar qué ocurre, incluir nuevas opciones, cambiar y añadir... No es fácil romper nada y, en tales casos, basta aplicar la regla de oro de la informática: cerrar `RStudio` y volver a abrirlo.

[4] Y cualquier otro, incluidos los humanos.

Ejercicio 2.1.

En `RStudio`, podemos situarnos en la ventana de archivo (`File/ New File/R Script`) y crear un archivo nuevo (yendo al menú o pulsando el icono situado más a la izquierda). Después escribimos el comentario `# Cap_02.R` en la primera línea; al dar a `Enter` vemos que el símbolo `#` aparece en la siguiente línea (luego será un comentario que tampoco se ejecutará).

Continuamos tal como aparece en la figura 2.1, incluyendo nuestro nombre. Después grabamos el archivo en el directorio que elijamos, nombrándolo `Cap_02.R`.

Después podemos ir al directorio elegido y abrir el archivo con un editor de textos cualquiera, para comprobar que sólo contiene texto común.

2.1.1 *Calculadora*

La forma más fácil de empezar es usando `R` como calculadora. Podemos escribir en la consola:

```
2 + 3
```

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

Y obtenemos el resultado esperado. Los espacios no son necesarios, pero se recomiendan para mejorar la legibilidad. Podemos usar igualmente operadores aritméticos y otros habituales. En el ejemplo realizamos algunas operaciones aritméticas, `(7 * (4 - 6.5))`, obtenemos su valor absoluto con la función `abs()` y calculamos la raíz cuadrada mediante `sqrt()`:

```
sqrt(abs(7 * (4 - 6.5)))
```

```
## [1] 4.1833
```

2.1.2 *Evaluaciones y asignaciones*

En cada ejemplo anterior se ha realizado una *evaluación*: un proceso por el que el intérprete del lenguaje `R` comprueba si la expresión es sintácticamente correcta y, en caso afirmativo, la ejecuta y devuelve un resultado. El resultado es volátil: se muestra en la consola pero no queda almacenado.

```
2 * 3
```

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

Si queremos almacenar el resultado hemos de hacer una *asignación* mediante el operador (`<-`) o bien mediante el signo igual, (`=`).[5] El

[5] Ambos sirven para asignar aunque hay alguna diferencia que ya veremos.

valor proporcionado se asignará a una *variable* que, si no existe previamente, se creará en el momento de la asignación.[6]

[6] Y si existe, su valor será sustituido (*machacado*) por el nuevo, *sin mediar ningún aviso*.

```
a <- 2 * 3
```

El operador de asignación se forma mediante los símbolos *menor que*, `<`, y *menos*, `-`, y puede utilizarse en ambos sentidos. La asignación anterior, por tanto, también puede expresarse como:

```
2 * 3 -> a
```

Se observa que las asignaciones no ofrecen ningún resultado como respuesta (aunque su valor aparece en el espacio de trabajo, en la ventana *Environment* de `RStudio`)[7]. Podemos consultar su contenido simplemente escribiendo su nombre, mediante impresión *implícita*, o bien mediante la función `print()`, de forma *explícita*.

[7] Otra forma de lograrlo es mediante el uso de paréntesis en una asignación:

```
(a <- 2 * 3)
## [1] 6
```

```
a
```

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

```
print(a)
```

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

En la consola, el resultado de ambos procedimientos es idéntico. Pero en el caso de ejecutarse desde un programa o función, sólo la segunda opción mostrará el contenido de los objetos.

Hay que tener presente que `R` es sensible a las mayúsculas y minúsculas, luego `a` y `A` son variables distintas. La función `objects()` muestra los objetos presentes en la memoria de trabajo.

```
A <- -10
a * A
```

```
## [1] -60
```

```
objects()
```

```
## [1] "a" "A"
```

2.1.3 *Nombrar*

Las variables se nombran utilizando letras, números, y los caracteres punto (`.`) y guión bajo (`_`). Los nombres pueden comenzar por una letra o un punto[8]. Las funciones siguen reglas similares, pero el punto tiene (o puede tener) un significado especial[9].

Tampoco pueden utilizarse las palabras reservadas, que podemos encontrar en la ayuda (o ejecutando `?reserved`) y que se presentan a continuación:

[8] Que no vaya seguido por un número: `.Valor` sería, por tanto, un nombre válido; `.1Valor`, no.

[9] En las funciones *genéricas*, donde el punto indica que esa función sólo se aplica a determinadas clases de objeto. Se verá en el capítulo 15, dedicado a las funciones.

```
if else repeat while function for in next break

TRUE FALSE NULL Inf NaN NA NA_integer_ NA_real_ NA_complex_
NA_character_
```

Así, son nombres válidos `A`, `d12`, `Valor`, `DATO`, `.B1` o `FOR`, y no lo son `1A`, `.6VAR`, `_DATO` o `for`.

Además de *nombrar*, también es conveniente *comentar* los programas, explicando qué se lleva a cabo en cada parte; ello ayuda a comprender posteriormente el código. Los comentarios se indican mediante el carácter *almohadilla* (`#`), y todo lo que aparezca posteriormente en la misma línea no será ejecutado.

```
# Los comentarios no se ejecutan
# print("No se ejecuta")
print("Sí se ejecuta")
```

```
## [1] "Sí se ejecuta"
```

Ejercicio 2.2.
Queremos calcular la circunferencia y área de un círculo de radio 1.8 metros. Recordemos que la circunferencia es dos *pi* veces el radio; el área, por su parte, es *pi* veces el cuadrado del radio.

Define y nombra las variables adecuadamente y realiza los cálculos necesarios.

2.2 Algunos objetos de R: vectores y funciones

Es frecuente oír que *todo lo que existe en R es un objeto*. Más allá de cuestiones terminológicas, el resultado práctico de este hecho es que, a cierto nivel, todo lo que forma parte de R es tratable de igual forma: todo objeto puede copiarse, duplicarse, almacenarse, asignarse...

Pues bien, de entre todos los *tipos* de objetos que existen en R[10], hay dos grupos especialmente interesantes: aquellos que sirven para *almacenar* información y los que permiten *procesarla*; nos referimos a *vectores* y *funciones*.

[10] Existen 24 tipos de objetos en R, como veremos a su debido tiempo.

Un aspecto importante a destacar es que en R, las variables creadas no necesitan ser *declaradas*, esto es, indicar su tipo (entero, doble, carácter...) El tipo será asignado automáticamente dependiendo del contenido del vector, como veremos.

2.2.1 Vectores

En R, todas las estructuras que almacenan datos son vectores. Ya se habrá observado que antes de las respuestas siempre aparece `[1]`. Efectivamente:

```
3 * 4
```

```
## [1] 12
```

La razón para ello es que el resultado no es un `12` sin más (esto es, un *escalar* con valor `12`); sino un *vector* de longitud unidad cuyo primer (y único) elemento es `12`. El número entre corchetes indica el índice correspondiente a ese primer elemento. Usamos el término *vector* en la acepción propia del ámbito informático: un conjunto ordenado de elementos a los que puede accederse mediante un índice.

Una particularidad de `R` es que es un *lenguaje vectorial*, de forma que, al operar sobre un vector, opera sobre *cada uno* de sus elementos, lo que simplifica enormemente el tratamiento de éstos.[11] Por lo demás, cuando tenemos vectores de longitud `1`, `R` funciona de forma equivalente al tratamiento de escalares.

[11] Lo que procede de su diseño enfocado a la estadística. A fin de cuentas, una variable tiene estructura vectorial de forma natural.

Si queremos construir un vector, la forma más simple es mediante la función `c()`, que concatena los elementos introducidos:

```
v <- c(9, -4, 3.14)
v
```

```
## [1]  9.00 -4.00  3.14
```

Una vez construido, el vector `v` está disponible en memoria. Si queremos acceder a un elemento particular, debemos introducir su índice entre corchetes:

```
v[3]
```

```
## [1] 3.14
```

Como veremos luego, existen vectores *atómicos* y *no atómicos*. Los primeros sólo pueden almacenar un mismo tipo de elementos (enteros, caracteres...); los segundos pueden almacenar elementos variados y son mucho más versátiles.

2.2.2 *Funciones*

También se puede oír que *toda acción realizada en `R` consiste en la ejecución de una función*. Y es cierto; la simple definición anterior implica la ejecución de dos funciones: *asignar* (`<-`) y *concatenar* (`c()`):

```
v <- c(9, -4, 3.14)
```

Si los vectores permiten almacenar los datos, las *funciones* operan con esos datos (entre otras muchas tareas). Las funciones van seguidas siempre de paréntesis entre los que se incluyen los *argumentos* que necesita[12].

[12] Algunas funciones no precisan argumentos, o pueden usar argumentos predefinidos, aunque sigue siendo necesario el uso de paréntesis. La siguiente función, por ejemplo, ofrece el directorio de trabajo actual:

```
getwd()
## [1]
'/Users/jose/Documents/FDR'
```

```
mean(v)      # Media aritmética
```

```
## [1] 2.713333
```

Vemos que `mean()` devuelve un único resultado al recibir un vector. Otras funciones, sin embargo, operan elemento a elemento, devolviendo como resultado un vector de longitud igual al introducido.

```
sqrt(v)      # Raíz cuadrada
```

```
## Warning in sqrt(v): Se han producido NaNs
```

```
## [1] 3.000000      NaN 1.772005
```

Se observa que ha habido problemas con el segundo elemento, `-4`, del vector `v` al intentar calcular su raíz cuadrada. `R` ha devuelto `NaN`, acrónimo de *Not a Number*, o indeterminación.

Existen multitud de funciones en la instalación base de `R`, y también en los *paquetes* disponibles. Y además, es posible construir nuestras propias funciones para realizar cualquier tipo de tarea. En el ejemplo construimos la función `raiz()`, que convierte la entrada en compleja si algún valor es negativo[13]:

[13] Solucionando así el problema anterior, ya que en el dominio complejo sí existe solución.

```
raiz <- function(entrada) {
    if(any(entrada < 0)) {
        entrada <- as.complex(entrada)
    }
    return(sqrt(entrada))
}
raiz(v)
```

```
## [1] 3.000000+0i 0.000000+2i 1.772005+0i
```

`R` es un lenguaje de alto nivel, lo que quiere decir que muchas de sus instrucciones expresan procesos complejos con un lenguaje próximo al lenguaje (inglés) natural. Así, es fácil entender qué realiza el código anterior: `raiz()` se define (`<-`) como una función (`function`) con el argumento `entrada`; si (`if`) algún (`any`) elemento de la `entrada` es menor que `0`, entonces `entrada` se definirá (`<-`) como complejo (`as.complex`); al final, se devuelve (`return`) la raíz cuadrada (`sqrt`) de la `entrada`.

Las funciones son también *objetos* y, como tales, las funciones creadas pueden verse dentro del entorno de trabajo, en la ventana *Environment* de `RStudio`. Así, si se ejecuta el código anterior, dispondremos de un nuevo objeto, `raiz()`, visible al mostrarlos con la función `ls()`[14].

[14] Equivalente a `objects()`.

```
ls()
```

```
## [1] "a"    "A"    "raiz" "v"
```

Más allá de hacer cálculos sobre los datos, las funciones abarcan todo tipo de tareas, como pueden ser:

- Control del entorno, directorio de trabajo, borrado de objetos...
- Entrada y salida de archivos, importación y exportación...
- Creación de objetos, selección, transformación...

Como se ha observado, y como norma a lo largo de todo el texto, las funciones siempre se mostrarán con los paréntesis, lo que ayuda a diferenciarlas del resto de objetos.

Ejercicio 2.3.
Queremos crear el vector `w`, que contiene los valores 1, -3, 7 y 12. Luego aplicaremos dos funciones al vector: una que dé un resultado único, de tipo resumen (como la media) y otra que dé un resultado elemento a elemento (de igual longitud). Después, comprueba los objetos presentes en el espacio de trabajo.

2.3 Más sobre vectores

Sea cual sea el uso que demos a `R`, todos los valores que necesitemos tratar serán almacenados en vectores. Tanto si definimos un simple valor como en `n = 20`, como si tratamos un archivo de 1500 casos por 300 variables, ambas informaciones se almacenarán en vectores.

Obviamente, el primer caso requerirá un tipo de vector mucho más simple que el segundo. Así, existe una división entre vectores *atómicos* y *no atómicos*. Los primeros contienen *un tipo* determinado de elementos (enteros, reales, caracteres...) *y sólo uno*; los segundos son más complejos, y consisten en agrupaciones de los primeros.

2.3.1 Vectores atómicos

Existen cinco tipos de vectores atómicos en `R`: enteros, reales, lógicos, alfanuméricos y complejos.[15] A continuación se define un vector de cada tipo:

[15] Hay un sexto tipo, los vectores *raw*, muy poco frecuentes y que no trataremos aquí.

```
i <- c(10L, 13L)    # integer
d <- c(1, .6, 3)    # double
l <- c(T, FALSE)    # logical
c <- c("A", "txt")  # character
x <- c(1+2i, 4i)    # complex
```

Cuando hablamos de cinco *tipos* no nos referimos una clasificación conceptual sino estructural; el *tipo* es una propiedad de los objetos[16], y muy importante: indica la estructura de almacenamiento utilizada lo que, en términos prácticos, significa que define todas sus propiedades en cuanto a lo que puede hacer y lo que no.[17]

[16] Que podemos obtener a través de la función `typeof()`.

[17] Véase que en el ejemplo se han incluido *varias* funciones en la misma línea, lo que puede hacerse usando el punto y coma, ';'.

```
typeof(i); typeof(d); typeof(l); typeof(c); typeof(x)
```

```
## [1] "integer"
```

```
## [1] "double"
```

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

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

```
## [1] "complex"
```

A continuación mostramos una breve descripción de cada tipo:

Enteros (`integer`). Almacenan valores numéricos únicamente enteros. En el ejemplo se ha utilizado el sufijo `L` para indicar que los valores numéricos son de tipo entero[18].

Reales (`double`). Almacenan valores numéricos reales[19]. Es el tipo por defecto, esto es, el asignado cuando no se indica nada (como se observa en el ejemplo).

Lógicos (`logical`). Sólo pueden almacenar valores *verdadero* y *falso*; se puede usar la palabra completa (`TRUE`, `FALSE`) o las iniciales (`T`, `F`), pero siempre en mayúsculas.

Alfanuméricos (`character`). Cualquier cadena de caracteres entrecomillada se considera de tipo alfanumérico; se pueden usar comillas `"dobles"` o `'simples'`.

Complejos (`complex`). En el ejemplo se definen usando la expresión algebraica como suma de una parte real y otra imaginaria (definida mediante el sufijo `i` unido, sin espacios, a algún valor numérico). Véase que tampoco se ha dejado espacio alrededor del signo `'+'`.

[18] La razón se remonta muchos años atrás, y se verá en el capítulo 5.

[19] La razón del nombre `double` también se verá en el mismo capítulo.

Hay otra forma habitual de generar (una secuencia de) enteros mediante el operador dos puntos (`:`); conviene conocerla porque es frecuente en multitud de operaciones, pero también en la descripción de los objetos. En el ejemplo, creamos un vector entero mediante una secuencia de `10` a `15`:

```
(i <- 10:15)
```

```
## [1] 10 11 12 13 14 15
```

Y también se utiliza a la hora de mostrar la estructura de los objetos, con la función `str()`:[20]

[20] Tanto la función `typeof()` como `str()` se verán en detalle en el siguiente capítulo. Respecto a `str()`, basta saber por ahora que muestra una descripción breve del objeto que, en este caso, incluye la longitud del vector expresado mediante el operador dos puntos: `[1:6]`.

```
str(i)
```

```
##  int [1:6] 10 11 12 13 14 15
```

Los vectores atómicos también pueden configurarse como estructuras bidimensionales (matrices) o de más dimensiones (*arrays*), pero lo dejamos para más adelante.

Recordemos ahora que estos vectores son objetos *atómicos*, lo que indica que sólo pueden contener un tipo de datos. ¿Qué ocurre, por tanto, si mezclamos elementos de diferente tipo en un mismo vector? ¿Obtendremos un error?

```
A <- c(3, "a"); A
```

```
## [1] "3" "a"
```

No hay error. `R` ha *coercionado* los elementos al tipo que permita almacenar ambos. En tanto no hay forma (unívoca) de convertir `'a'` en un número, el valor real `3` es convertido al tipo `character`.

¿Cómo tratar, entonces, con un archivo de datos, que puede contener información de varios tipos? Necesitamos estructuras de almacenamiento no atómicas.

Ejercicio 2.4.

Vamos a crear un vector atómico de cada uno de los cinco tipos vistos hasta ahora, con dos condiciones: primera, se crearan en orden, empezando por el tipo que menos versatilidad tiene y acabando con el que más; segundo, han de tener longitud creciente, comenzando en dos.

Consulta el tipo y la estructura de cada vector creado.

2.3.2 *Vectores no atómicos*

Hay dos objetos que permiten almacenar información heterogénea: las listas y los *data frames*[21].

Listas. Se les llama a veces *contenedores universales*, ya que pueden almacenar cualquier vector atómico e incluso también listas y *data frames*, y en cualquier cantidad. Son muy utilizadas para *reunir* información relacionada (por el ejemplo, los múltiples resultados de un análisis estadístico: coeficientes, descriptivos, intervalos de confianza, los mismos datos...)

Data frames. Son listas especializadas, diseñadas para almacenar bases de datos en formato filas (casos) por columnas (variables).

Se construyen con las funciones del mismo nombre. En este caso creamos una lista con tres vectores de diferentes tipos: `double`, `character` y `logical`.

[21] No existe un término consensuado en castellano para *data frame*, aunque en el manual de *Introducción a R* lo vemos traducido como *hojas de datos*. Aquí hemos preferido dejarlo tal cual.

```
lista <- list(A = 7,
              B = c("uno", "dos"),
              C = T)
lista
```

```
## $A
## [1] 7
##
## $B
## [1] "uno" "dos"
##
## $C
## [1] TRUE
```

Con *data frames* el funcionamiento es similar, pero es requisito que los vectores componentes tengan la misma longitud, algo razonable en cualquier base de datos.

```
df <- data.frame(Id = c(1, 2, 4, 5),
                 Grupo = c("Exp", "Ctr", "Ctr", "Exp"),
                 VD = c(12, 11, 9, 14))
df
```

```
##   Id Grupo VD
## 1  1   Exp 12
## 2  2   Ctr 11
## 3  4   Ctr  9
## 4  5   Exp 14
```

Para acceder a su contenido existen varias formas; una de ellas utiliza el operador `$` para unir el nombre del objeto con el nombre del elemento.[22]

[22] Por supuesto, también puede accederse mediante corchetes '`[]`' como con cualquier otro vector. Lo veremos en los capítulos dedicados a estos objetos.

```
lista$B
```

```
## [1] "uno" "dos"
```

```
df$Grupo
```

```
## [1] "Exp" "Ctr" "Ctr" "Exp"
```

Los *data frames*, además, ofrecen un acceso de tipo matricial indicando la fila y columna mediante un par de índices. En los ejemplos se seleccionan un caso, una variable y la intersección de ambos[23].

[23] Véase que dejar un hueco se interpreta como *todos los casos* o *todas las variables* según esté en la primera o segunda posición, respectivamente.

```
df[2, ]   # Caso 2, todas las variables
```

```
##   Id Grupo VD
## 2  2   Ctr 11
```

```
df[, 3]   # Variable 3, todos los casos
```

```
## [1] 12 11  9 14
```

```
df[2, 3]  # Caso 2, variable 3
```

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

Ejercicio 2.5.
Utilizando los vectores construidos en el ejercicio anterior, crea una lista con tres componentes. Contruye también un *data frame* con los dos vectores que no usaste para la lista. Probablemente tendrás que hacer algún ajuste respecto a la longitud de los componentes.

Prueba a crear otras listas y *data frames* con otras combinaciones de esos mismos componentes.

2.3.3 *R, un lenguaje vectorial*

El carácter vectorial de R no se limita al hecho de que todas sus estructuras de almacenamiento de datos sean vectores. De hecho, estas estructuras no tendrían mayor valor en sí mismas si no estuviesen acompañadas de un *procesamiento vectorial*.

De esta forma, muchas de las acciones que en otros lenguajes requieren operar sobre *cada elemento* de un vector mediante algún tipo de bucle repetitivo, en R se realizan directamente.

Y no es casualidad. Recordemos que R nace orientado al análisis estadístico, donde el tratar con variables constituye el modo habitual de trabajo. Y una variable se representa de forma natural como un conjunto de valores ordenados (según el índice de cada caso); precisamente un vector.

Por ejemplo, si quiero multiplicar *todos* los elementos de un vector por una cantidad, no necesito multiplicar *cada elemento* por esa cantidad; basta multiplicar el vector como un todo:

```
v <- 1:5
v * 10
```

```
## [1] 10 20 30 40 50
```

Si volvemos a la estadística, una operación como calcular una media o una varianza, que suele requerir recorrer cada elemento una y otra vez, en R se reduce a expresiones simples que no necesitan de ningún tipo de bucles.

```
media <- sum(v)/length(v)
varianza <- sum((v - media)^2)/(length(v) - 1)
```

Este modo de operar requiere un modo de pensar algo diferente, *vectorial*, y es frecuente que se tarde un tiempo en lograrlo. Y la vía más directa es, sin duda, la práctica.

Ejercicio 2.6.
Obtén el vector `z`, resultado de tipificar el vector `v`, usando procedimientos vectoriales.

2.4 Lectura y escritura de archivos

Como cualquier otro lenguaje, `R` puede escribir y leer diferentes tipos de información en múltiples formatos. Pero dentro de esta gran variedad, hay dos tipos principales de archivos que todos usamos: archivos de programa y archivos de datos.

Archivos de programa. También llamados *programas* sin más, no son más que una secuencia de acciones (instrucciones en lenguaje `R`), agrupadas en un archivo con la extensión `.R`. El archivo `Cap_02.R`, en el que íbamos almacenando las instrucciones de este capítulo, es un ejemplo de archivo de programa.

Archivos de datos. Son los que contienen la información *procedente de*, por ejemplo, una investigación, y que podemos leer desde `R`. También pueden ser datos *generados por* `R`, por ejemplo los resultados de una simulación que almacenamos para ser analizados posteriormente. En `R`, los archivos de datos tienen la extensión `.RData`[24], como en `datos.RData`.

[24] O simplemente `.rda`.

2.4.1 Archivos de programa

Un archivo de programa no es más que un archivo de texto con extensión `.R` que contiene código (instrucciones) de `R`. Podemos escribirlo en cualquier editor de texto, aunque siempre será mejor en una interfaz de `R`, para poder ir ejecutando y probando el código conforme se escribe[25].

[25] En las interfaces especializadas como `RStudio`, además, colorean el código de forma que hacen más reconocibles las funciones, variables, etc.

Un ejemplo de archivo de programa es `Cap_02.R`, el archivo que hemos sugerido crear con los ejemplos del libro. Al usarlo, comprobamos que la manera natural de trabajar es recorrer las líneas y ejecutar aquéllas que nos interesen; a veces se copia y pega una instrucción, se modifica y se ejecuta para observar su comportamiento. Aunque el objetivo difiere, la forma del proceso no dista mucho de la utilizado en un análisis de datos.

A la vez, en informática es frecuente llamar *programa* a un conjunto de código *cerrado* que realiza siempre una misma acción cuando

se ejecuta[26]. Pero recordemos que R es, por diseño, un lenguaje interpretado (esto es, que no genera archivos que puedan ejecutarse con independencia del entorno de R), orientado al análisis estadístico y, con esta lógica, pensado para aplicar unos procedimientos a unos datos cuyos resultados servirán de entrada a otros procedimientos.

En resumen: en el trabajo con R es habitual utilizar archivos de programa, pero haciendo una ejecución interactiva del código, tal como hacemos en `Cap_02.R`. Supongamos, por ejemplo, un análisis estadístico básico; lo habitual es almacenar en un programa los pasos a dar, que suelen ser:

- Lectura o importación de los datos y almacenamiento en un *data frame*.
- Análisis inicial, corrección de errores, depuración y grabación del resultado en un nuevo archivo de datos.
- Descriptivos, gráficos, supuestos...
- El análisis en sí.
- Posibles análisis posteriores.

Cada paso es programado y ejecutado, se estudia el resultado y se decide el próximo paso, que es programado y ejecutado... Almacenar los pasos permite repetir los procedimientos, corregir errores, afinar los análisis y extraer los resultados de diferentes maneras. Incluso cuando todo ha finalizado, el archivo guardado puede retomarse como base para otro análisis.

Por supuesto, también es posible tener programas que sean *cerrados*, pero suelen ser más simples y unívocos que lo que se requiere, por ejemplo, en un análisis de datos completo. Un caso sería el que ha llegado a ser el primer ejemplo en cualquier lenguaje de programación: un programa que, al ser ejecutado, muestra por pantalla el mensaje *"¡Hola, mundo!"*.

Para ello basta escribir la instrucción `print("¡Hola, Mundo!")`, almacenarla en un archivo llamado `hola.R` y ya tenemos todo un programa, completamente funcional.

```
print("¡Hola, Mundo!")
```

Podemos ejecutarlo, desde consola o desde dentro de otro programa, mediante la instrucción `source()`.

```
source("hola.R")
```

```
## [1] "¡Hola, Mundo!"
```

Más allá de la simpleza del ejemplo, los programas ejecutados de esta forma habrán de ser, necesariamente, más simples que lo requerido por cualquier análisis[27].

[26] Y que generalmente tiene una versión ejecutable (por ejemplo, un archivo del tipo `.exe` o `.app`) que puede ejecutarse con independencia del programa que lo creó.

[27] En principio, aunque es posible que el programa sea interactivo, y solicite, por ejemplo, el nombre del archivo de datos a utilizar o permita hacer elecciones.

Ejercicio 2.7.
Vamos a crear un pequeño programa y luego lo ejecutaremos. Para ello:

- Abre un archivo nuevo (*R Script*) en `RStudio`.
- Escribe un comentario (con #) al inicio indicando el nombre del programa (`programa_1.R`) y en la siguiente línea, su autor/a.
- A continuación crea un vector, v, de longitud 7 con números reales en el intervalo $[0, 10]$.
- Calcula la media del vector.
- Muéstrala por pantalla.
- Guarda el archivo con el nombre indicado anteriormente y ciérralo.
- Ejecuta el programa.

2.4.2 *Archivos de datos*

Más frecuente aún es la creación y lectura de *archivos de datos*, que en `R` tienen la extensión `.RData`. Por ejemplo, si trabajando con `R` hemos construido el *data frame* `df_datos` con los datos ya depurados, podemos almacenarlo en un archivo usando la función `save()` como en el ejemplo, lo que creará el archivo `datos.RData`.

```
save(df_datos, file = "datos.RData")
```

De igual forma, en la mayoría de los casos los datos *son leídos* de un archivo de datos, procedente de `R` (en formato `RData`), como en el ejemplo a continuación, mediante la función `load()`.

```
load("datos.RData")
```

Pero otras muchas veces el archivo se ha construido con un programa más cómodo para el registro de datos[28]. Es por ello que `R` incluye funciones para importar datos de (y exportarlos a) otros formatos. Éstos incluyen archivos de texto (con variadas opciones para separadores, decimales, etc.), hojas de cálculo, archivos de otros programas estadísticos, bases de datos, etc.

A pesar de su utilidad, el capítulo dedicado a la lectura y escritura de archivos queda aún lejos (en el capítulo 14). Antes necesitamos conocer, en los capítulos previos, las estructuras de almacenamiento que luego albergarán la información procedente de esos archivos.

[28] Por ejemplo, una hoja de cálculo. En cualquier caso, no es frecuente construir *data frames* manualmente, tal como hicimos en los ejemplos anteriores, escribiendo las puntuaciones para cada vector.

Ejercicio 2.8.
Guarda el anterior vector `v` en un archivo llamado `datos.RData` en el directorio de ejercicios. Comprueba la existencia del archivo en el directorio donde se ha almacenado.

Ejercicio 2.9.

Ahora vamos a construir otro programa, `programa_2.R`, que leerá el archivo de datos recién construido (`datos.RData`) y calculará y mostrará la media. Para ello:

- Abre el archivo `programa_1.R` en `RStudio`.
- Guárdalo con el nombre `programa_2.R`.
- Cambia la línea donde se creaba el vector `v` por otra donde se lea el archivo de datos `datos.RData`.
- Deja el resto del programa igual.
- Guarda el archivo y ciérralo.
- Comprueba que el archivo existe y está en el mismo directorio en que estamos situados en `R`.
- Ejecuta el programa.

Ejercicio 2.10.

Por último, modificaremos el vector `v`, lo volveremos a grabar en `datos.RData` y ejecutaremos de nuevo `programa_2.R`. También deberemos comprobar que el resultado de la ejecución se ajusta al nuevo vector.

2.5 *Paquetes*

Un paquete (*package*) es un conjunto de funciones que amplían las capacidades de `R`. Como ya se dijo, los paquetes son una de las grandes fortalezas de `R`[29], tanto por su cantidad y calidad como por la velocidad de aparición y actualización.

Para poder usar las funciones de un paquete hemos de dar tres pasos previamente: descargar el paquete, instalarlo y cargarlo en memoria. Todo ello puede hacerse mediante la interfaz de `RStudio`, o también mediante código. Por ejemplo, si queremos calcular un estadístico de asimetría (no disponible en la instalación base de `R`) obtendremos un error, ya que no existe tal función en la distribución base de `R`.

[29] Llegando a los 15000 paquetes para finales de 2019 y superando los 18000 en 2021, como se observa en la sección *Packages* en la *web* del proyecto `R` (`https://cran.r-project.org/web/packages/`.

```
v <- c(2, 4, 3, 2, 5, 4, 3, 2)
skewness(v)               # Intenta aplicar una función
```

```
## Error in skewness(v): no se pudo encontrar la función "skewness"
```

Podemos usar la función `skewness()`, disponible en el paquete `e1071`. Para ello descargamos, instalamos y cargamos en memoria dicho paquete.

```
install.packages("e1071")  # Descarga e instala
library("e1071")           # Carga en memoria
```

Ahora sí es posible aplicar la función.

```
skewness(v)                # Aplica una función
```

```
## [1] 0.3201403
```

Cualquier usuario, además, puede construir sus propios paquetes, bien para uso personal, bien para ponerlos a disposición de otros en la CRAN (o en otros repositorios como, por ejemplo, GitHub[30]).

[30] https://github.com/features.

Ejercicio 2.11.
Los vectores `x <- c(2, 1, 5, 2, 4)` e `y <- c(1, 1, 4, 3, 4)` contienen información ordinal. Queremos conocer una medida de su relación a través de la correlación policórica.
Comprueba si R dispone de alguna función que responda a nuestra necesidad; en caso contrario, localiza algún paquete que sí permita el cálculo, instálalo y llévalo a cabo.

2.6 *La ayuda*

Además de todos los recursos disponibles en la *web*[31], R posee un sistema de ayuda que nos informa en detalle sobre cada instrucción del lenguaje R. El sistema está disponible sin necesitar conexión a la red, y contiene información sobre las instrucciones de *todos* los paquetes instalados en nuestro ordenador.

[31] Y que ya indicamos en el capítulo anterior.

Es importante notar cómo, a pesar del carácter voluntario y colaborativo de las contribuciones en R, existe una estructura rígida a la hora de construir los paquetes que proporciona una uniformidad muy deseable. Uno de los resultados visibles se encuentra en la ayuda, que siempre muestra la misma estructura e información (descripción, uso, argumentos, detalles, valores...) Teniendo en cuenta la necesidad de consultar frecuentemente la ayuda para todo programador, esa uniformidad facilita la tarea en gran medida.

Existen diferentes formas de acceder a la ayuda. Una de ellas es escribir `help.start()`, o bien ir al menú `Help/R Help`[32], pero la forma más común es acudiendo a la ventana de ayuda de RStudio (figura 2.2).

[32] Tanto en la interfaz básica de R como en RStudio, aunque los resultados difieren a veces en algunos aspectos.

Hay otras formas de obtener ayuda, como escribir `help("mean")` para acceder a información sobre alguna instrucción particular, o bien usando un signo de interrogación seguido del nombre del comando deseado, `?mean`, lo que nos llevaría a la misma ventana de ayuda de la figura 2.2. Cuando no buscamos un comando de R, sino algún término de interés (por ejemplo, para saber cómo dibujar una elipse), podemos usar `help.search('ellipse')` o bien una doble interrogación seguida de la palabra a localizar (`??ellipse`). R buscará ese término en cualquier lugar de la ayuda disponible, lo que nos llevará de nuevo a la ventana de ayuda, mostrando todos los casos en que aparece.

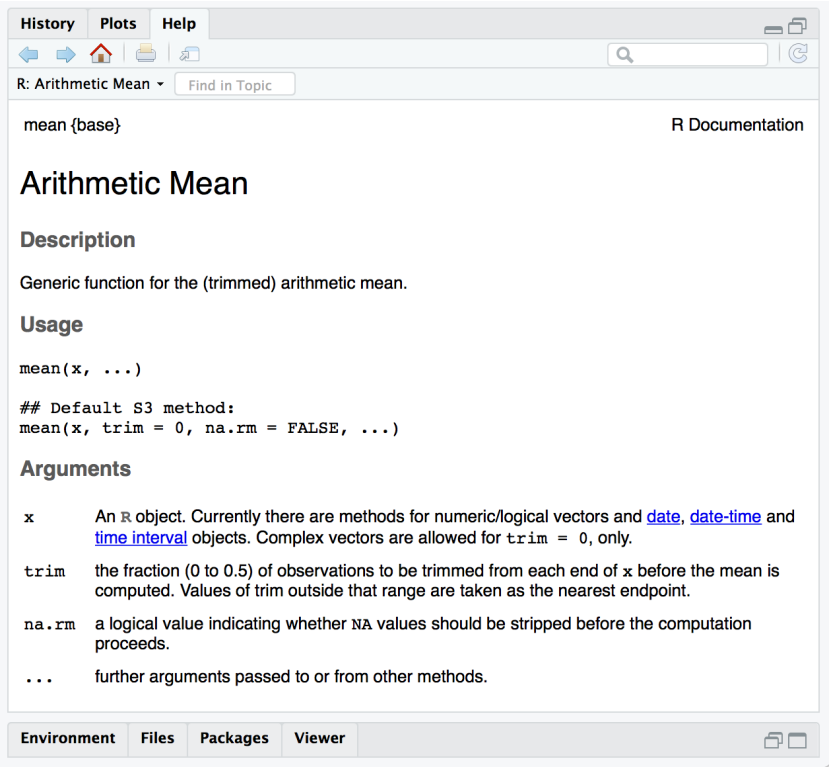

Figura 2.2: Ventana de ayuda en `RStudio` para la función `mean()`.

Dos funciones más complementan la ayuda disponible: `example()` muestra ejemplos de uso de la función indicada (extraídos de la ayuda de la función), mientras que `demo()` hace demostraciones de las capacidades de R. Algunos ejemplos pueden ser los siguientes:

```
?plot
??png
example(apply)
demo(colors)
```

Ejercicio 2.12.
¿Cómo obtener una media recortada?
Una búsqueda en la ayuda (`?trimmed mean`) no dice nada, y usar la doble interrogación tampoco ayuda mucho. Una búsqueda en Internet puede ayudar, y también explorar la ayuda de la función `mean()`.

2.7 *Salir de R*

Al acabar una sesión, indicamos a R que queremos salir con la función `q()`, del inglés *quit*. Si existe algún archivo de código sin grabar nos preguntará si queremos guardarlo a través de una ventana de diálogo.

```
q()
```

```
## Save workspace image to ~/Documents/B/.RData? [y/n/c]:
```

Como se aprecia, también nos consulta acerca de guardar el espacio de trabajo, esto es, los objetos activos en memoria, y que podemos almacenar en un archivo para tenerlos disponibles al abrir otra sesión[33].

Deberemos responder indicando la inicial de **y**es, **n**o o **c**ancel. Los detalles de los procesos realizados al cerrar (y al abrir) R se verán con detalle en el capítulo 20. En general, no se recomienda, ya que puede haber objetos presentes que no estén en una situación de uso normal, o que modifiquen el funcionamiento del programa.

[33] Es decir, que al volver a abrir tendremos los mismos objetos en memoria que en el momento de salir, con lo que podemos continuar el trabajo en el punto en que lo dejamos.

PARTE II: Fundamentos

3
Control del entorno y objetos (I)

El trabajo diario con R exige un control adecuado de lo que ocurre en cualquier sesión, y ello necesita de algunos conocimientos básicos. Comenzaremos ahora por conocer algo más del entorno en que se ejecuta y de los objetos que se utilizan y dejaremos otras cuestiones más avanzadas para el capítulo 20.

RESUMEN

Localización (3.1). Veremos cómo saber el directorio en que estamos trabajando y el modo de moverse por ellos.

Espacio de trabajo (3.2). Trabajar en `R` requiere crear objetos y manipularlos. La ventana `Environment` en `RStudio` nos proporciona esa información.

Todo lo que existe en R es un objeto (3.3). Desde los valores hasta las funciones, todo es un objeto almacenable y procesable. Veremos qué tipos de objetos básicos hay y cómo conocer sus propiedades.

Funciones genéricas (3.4). Son funciones, muy habituales, que se comportan de diferente manera según la propiedad `class` de objeto al que se aplican.

Operadores (3.5). Descripción básica y precedencia.

Avisos y errores (3.6). Parte imprescindible de aprendizaje, conviene conocer desde ya sus diferencias.

3.1 Localización

Sea uno consciente o no, siempre que ejecutamos un programa nos encontramos situados en algún directorio[1]. El resultado es que cualquier lectura o escritura de archivos será realizada, por defecto, en el directorio donde estamos situados.

Para conocer el directorio de trabajo (*working directory*) actual basta usar la función `getwd()`, y para cambiar de directorio usaremos `setwd()`, indicando entre los paréntesis, entrecomillado, el directorio

[1] Que no es lo mismo que una *carpeta*, aunque a veces coincidan. Por ejemplo, existe una carpeta llamada `Escritorio`, pero no existe un directorio con tal nombre; el directorio correspondiente se llama `Desktop`.

de trabajo que queremos establecer. Comenzamos consultando el directorio de trabajo actual.[2].

[2] La salida presentada es la obtenida en macOS.

```
getwd()
```

```
## [1] "/Users/jose/Documents/B"
```

El resultado es una cadena de caracteres con la ruta del directorio donde estamos situados. Dada la estructura de directorios de la figura de la derecha, si estamos en `B` y ahora queremos situarnos en `B1`, la función `setwd()` admite dos formas de indicar la ruta deseada :

Figura 3.1: Estructura de directorios.

1. Mediante una *ruta absoluta*: contiene la ruta completa desde el directorio raíz y, dada una estructura de directorios estable, podemos indicar cualquier directorio existente y siempre funciona.
2. Mediante *ruta relativa*: sólo se especifican los directorios *a partir de aquél en que estamos situados*. Por ese motivo, sólo funcionará cuando la posición *desde donde* se ejecute la función sea la adecuada.

A continuación vemos ambas formas de acceder al directorio `B1`.

```
setwd("/Users/jose/Documents/B/B1")  # absoluta
setwd("B1")                          # relativa
```

Las rutas absolutas tienen otra ventaja: si ahora quiero establecer como directorio de trabajo `B2`, no tengo más que indicarlo:

```
setwd("/Users/jose/Documents/B/B2")
```

Las rutas relativas, en cambio, han de partir del directorio de trabajo actual hasta llegar al deseado. En nuestro caso, si estamos situados en `B1`, debemos *volver* a `B` y luego *ir* a `B2`. Para ir hacia atrás (al directorio `B`, *padre* del actual, `B1`) usamos dos puntos (`..`), como vemos:

```
setwd("../B2")
getwd()
```

```
## "/Users/jose/Documents/B/B2"
```

¿Tienen, entonces, alguna utilidad las rutas relativas? Cuando la estructura de datos *no es estable* (por ejemplo, cuando copiamos unos programas a otro directorio o incluso a otro ordenador), todas la rutas absolutas tendrán que rehacerse. Pero si hemos utilizado rutas relativas (desde el directorio del programa, por ejemplo), todo seguirá funcionando igual.

En `RStudio` podemos ver el directorio de trabajo en la cabecera de la ventana de consola (véase la figura 3.2). La virgulilla (~) indica el directorio `Home`, o directorio del usuario desde cuya cuenta se ha

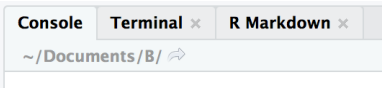

Figura 3.2: Directorio de trabajo, en la ventana de consola. Pulsando la flecha se actualiza la pestaña de directorios de `RStudio`.

iniciado R (en nuestro caso, `/Users/jose/`, y es otra forma completamente válida de indicar rutas en R). Junto al directorio, a su derecha, vemos una flecha apuntando también a la derecha; si la pulsamos se activa la pestaña `Files` en `RStudio` y se muestra el directorio de trabajo con todo su contenido (figura 3.3). Desde ahí podemos abrir en `RStudio` cualquier archivo que deseemos.

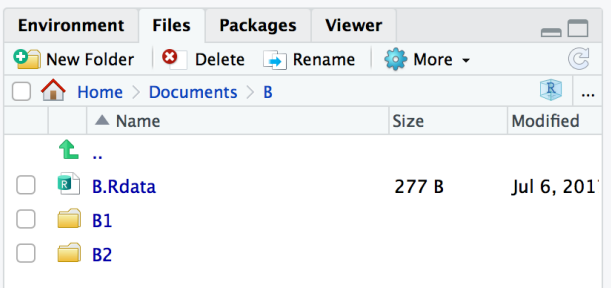

Figura 3.3: En `RStudio`, la ventana `Files`, mostrando los directorios utilizados en el ejemplo anterior.

En resumen, es imprescindible saber en cada momento cuál es nuestro directorio de trabajo, así como la ubicación de nuestros archivos de lectura y/o escritura, y volveremos sobre ello en el capítulo sobre entrada y salida de información.[3].

En ese sentido, el siguiente ejercicio es, más que recomendable, casi imprescindible para seguir ordenadamente el resto del manual.[4]

Ejercicio 3.1.

Es de vital importancia tener un control adecuado sobre *dónde estamos* (desde dónde se está ejecutando R) y *dónde están nuestras cosas* (programas, datos, salidas...). Por ello, en este ejercicio trataremos ambas cuestiones.

Respecto a *dónde están nuestras cosas*, y si no lo has hecho ya, es un buen momento para tomar el control de los directorios (carpetas) y archivos. Usa tu explorador de archivos para localizar el lugar donde se alojarán *todos* los archivos (programas, datos, gráficos, etc.) que se irán generando al leer (y practicar) este manual.

Lo habitual es crear un directorio específico; aquí usaremos `FDR` (por Fundamentos de R, obviamente). Podemos situar el directorio colgando directamente de la carpeta `Documentos`, que corresponde al directorio `Documents` (en macOS; en Windows tenemos la carpeta `Mis Documentos`, que corresponde al directorio del mismo nombre: `Documents`) o en cualquier otro lugar que queramos. Posteriormente podremos crear nuevas carpetas (por ejemplo, para almacenar los datos; o una carpeta por capítulo), pero ahora ya tenemos un lugar donde se almacenará lo que vayamos construyendo.

Respecto a *dónde estamos*, esto es, desde dónde se está ejecutando R, depende de cada caso. En general, al abrir `RStudio` nos

[3] Y de nuevo, de un modo más técnico, en el apartado 20.3.

[4] Si las explicaciones del ejercicio no son suficientes, o si que quiere saber más sobre los proyectos, podemos acudir al apartado 20.4 del segundo capítulo dedicado al control del entorno.

situamos en el último directorio donde estábamos al cerrar la última vez. Podemos cambiar el directorio desde la pestaña `Files`, pero vamos a utilizar una herramienta de `RStudio` que puede facilitarnos las cosas: los *proyectos*.

Por ahora, basta decir que un *proyecto* es una forma de agrupar, organizar un conjunto de archivos (programas, datos, salidas...) de forma cómoda. Una de las comodidades tiene que ver con el *dónde estamos*, ya que el proyecto *siempre* se abrirá en el directorio desde el que se creó.

En resumen: si ya hemos creado nuestro directorio `~/Documents/FDR` podemos crear un proyecto en dicho directorio. Para ello vamos al menú `File` en `RStudio` y pulsamos en `New Project`; elegimos *Existing Directory* y lo indicamos. Se creará un archivo con e nombre del directorio y con extensión `.Rproj`; en nuestro caso, `FDR.Rproj`.

Una vez creado, podemos acceder pulsando en este archivo, y `RStudio` se abrirá en el directorio adecuado y mostrará el mismo estado en que se cerró la sesión anterior, espacio de trabajo y archivos abiertos incluidos.

3.2 *El espacio de trabajo o entorno global*

Podemos aplicar la pregunta del apartado anterior a los objetos que creamos en `R`: ¿dónde están situados? La respuesta es que todos los objetos presentes en una sesión de `R` (cada variable, *data frame* o función creada) están almacenados en el *espacio de trabajo*, llamado también *entorno global* (*Global Environment*). Más técnicamente, este espacio de trabajo constituye uno de los varios *entornos* (*environments*), o espacios de memoria de que dispone `R` para almacenar variables, funciones y datos.

Podemos observar los objetos activos (accesibles) en la sesión actual en la pestaña `Environment` en `RStudio` junto con algunas de sus propiedades (ver figura 3.4).

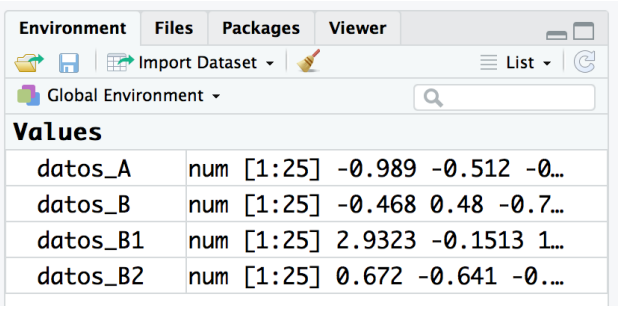

Figura 3.4: En `RStudio`, la ventana `Environment` muestra los objetos presentes en el espacio de trabajo. Además del nombre y parte de su contenido, se muestra su `clase` (`num`, que corresponde al tipo `double`) y su longitud (`[1:25]`).

También podemos saber qué objetos tenemos en nuestro espacio de trabajo usando cualquiera de las funciones `ls()` u `objects()`.

```
ls()
```

```
## [1] "datos_A"  "datos_B"  "datos_B1" "datos_B2"
```

Existen otros entornos, por ejemplo los asociados a los paquetes cargados en memoria, y que proporcionan un espacio independiente para almacenar sus variables y funciones (véase la figura 3.5), pero se tratarán brevemente en el capítulo 18, dedicado a los paquetes.

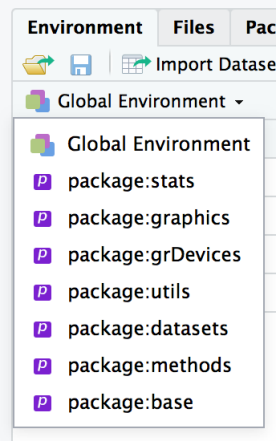

Figura 3.5: Entornos, o *environments* disponibles en una sesión básica de R.

Ejercicio 3.2.

Vamos a comprobar la utilidad de los *proyectos*. Si estamos en `RStudio` con uno o más archivos abiertos y con algunos objetos en el espacio de trabajo, podemos hacer lo siguiente: primero, cerramos `RStudio`, y damos `OK` cuando se nos pregunte si guardar el espacio de trabajo; a continuación vamos al directorio que contiene nuestro archivo `.Rproj` y lo ejecutamos.

Si todo es correcto, `RStudio` se abrirá en el punto exacto donde acabamos de cerrar, incluido el espacio de trabajo con todos los objetos.

3.3 Objetos y sus características

Decíamos que *todo lo que existe en `R` es un objeto*. Variables, funciones y otros elementos que veremos más adelante son objetos y, como tales, podemos consultarlos, almacenarlos, operar con ellos, visualizarlos...

En tanto el trabajo con `R` consiste en una continua manipulación de objetos, es necesario conocer sus características. Para ello disponemos de un conjunto de funciones cuya misión es informar de ciertos aspectos fundamentales de los objetos. Dos de estos aspectos son especialmente importantes: el *tipo* y la *clase*. Del primero hablamos en el capítulo anterior y ahora lo completamos; del segundo hablaremos ahora, junto con las demás características.

3.3.1 La importancia de conocer a fondo los objetos

Antes de comenzar es necesario insistir en la importancia de este apartado, aunque probablemente no se aprecie completamente hasta llevar leídos[5] unos dos tercios del manual. Y sin embargo, su defensa es bien simple: *todo lo que puede hacerse en `R` tiene la forma*

```
función(objeto),
```

luego el control de cualquier programa no consiste más que en tener clara conciencia, en cada línea, de las propiedades de la `función()` y del `objeto` (u `objetos`) que introducimos como argumento. Expresado de otra forma: cada vez que que tengamos un problema, su

[5] Y practicados.

solución pasará por el análisis de las propiedades de la función (sus argumentos de entrada, restricciones, salida) y del objeto (tipo, clase, atributos...)

A esto convendría añadir que R es un lenguaje extraordinariamente rico y versátil, lo que también puede traducirse, para quien no lo conoce, como desconcertante y propenso a errores. Es frecuente que existan varias maneras *correctas* de realizar una misma acción, por no hablar de ciertas operaciones automáticas[6] frecuentes que suelen desconcertar las primeras veces.

[6] Como el *reciclado* o la *coerción*.

En resumen, si queremos disfrutar de las bondades de R con seguridad o, lo que es lo mismo, control, todo nos lleva a conocer los objetos en profundidad.

Ejercicio 3.3.

Supongamos que tenemos tres vectores de igual longitud, X, Y y Z, y queremos obtener la matriz de correlaciones. Nada más fácil: sabiendo que `cor()` proporciona lo que queremos, vamos a la consola y escribimos:

```
cor(X, Y, Z)
```

```
## Error in cor(X, Y, Z): invalid 'use' argument
```

¿Qué ha ocurrido? ¿Qué ha fallado? ¿Qué significa la salida de error?

Y sobre todo, ¿cómo lo solucionamos? Todo pasa por atender a la ayuda de la función `cor()` y comprobar qué se requiere y qué hemos introducido.

3.3.2 *El tipo, `typeof()`*

La función `typeof()` devuelve el *tipo* del objeto, que será una de las 24 estructuras de almacenamiento disponibles en R. Como dijimos, el tipo determina la naturaleza del contenido y, derivadas de ella, sus propiedades.

Hasta ahora hemos visto seis *tipos* de objetos: cinco correspondientes a vectores atómicos (`integer`, `double`, `complex`, `character` y `logical`) y uno para vectores no atómicos (`list`, que es el tipo de listas y *data frames*).

Volveremos a los ejemplos del capítulo anterior para ilustrar estas funciones.

```
i <- c(10L, 13L)                    # integer
d <- c(1, .6, 3)                    # double
l <- c(T, FALSE)                    # logical
```

```
c <- c("A", "txt")                  # character
x <- c(1+2i, 4i)                    # complex
lst <- list(A = 7,                  # list
            B = c("uno", "dos"),
            C = T)
df <- data.frame(Id = c(1, 2, 4, 5),   # list
                 Grupo = c("Exp", "Ctr", "Ctr", "Exp"),
                 VD = c(12, 11, 9, 14))
```

Y aplicamos la función `typeof()` a algunos objetos.

```
typeof(c)
```

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

```
typeof(lst)
```

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

```
typeof(df)
```

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

En tanto todos los objetos necesitan ser almacenados, todos tendrán un tipo. Las funciones, por ejemplo, tienen *tres* posibles tipos, de los que dependen sus propiedades.[7]

[7] Como es el lenguaje en que está escrita la función (`R` o `C`) o el control de los argumentos. Pero todo esto se verá a su debido tiempo.

3.3.3 *La clase,* `class()`

La función `class()` sirve para informar de, y establecer, la *clase* de un objeto. Si el tipo era un atributo *estructural* que definía la naturaleza del objeto, la clase es un atributo *funcional*, que define los procedimientos que se podrán aplicar.

Su funcionamiento se basa en la programación orientada a objetos[8], de modo que ciertas funciones se comportan de diferente manera *dependiendo de la clase del objeto* al que se aplican.[9]

Al igual que ocurre con el tipo, *todos los objetos tienen una clase*. No obstante, conviene diferenciar entre dos formatos en que nos encontraremos las clases:

[8] Que veremos en detalle en el capítulo 21.

[9] En realidad esas funciones, llamadas *funciones genéricas*, se componen de múltiples *versiones* (técnicamente, *métodos*), uno para cada clase. Como otras cuestiones que ahora simplemente se presentan, se verán con todo detalle en su momento.

Clase implícita. Derivada del *tipo* de objeto, coincide con éste para los objetos que son vectores atómicos o listas. Una excepción es el caso de los vectores de tipo `double`, donde la clase es `numeric`.

Clase asignada (o explícita). Es la que se añade a un objeto[10], de forma que a partir de ese momento se le podrán aplicar todos los procedimientos (métodos) que existan para esa clase.

[10] Decimos *se añade* porque todos los objetos siguen manteniendo su clase implícita.

En general no tendremos que preocuparnos por la clase de un objeto, ya que `R` asigna las clases requeridas en su momento y aplica

los métodos adecuados. Ahora sólo veremos, a modo de ilustración, la clase de tres objetos: dos conocidos (un vector de enteros y un *data frame*) y uno nuevo (una matriz).

Podemos crear un vector de enteros mediante el operador ':' y su tipo y clase coincidirán: `integer`.

```
i <- 1:6
typeof(i); class(i)
```

```
## [1] "integer"
```

```
## [1] "integer"
```

Vemos que el vector ha heredado la clase del tipo; si usamos la función `attributes()`, que veremos a continuación, no muestra ningún atributo (`NULL`), ni siquiera la clase, ya que ésta no es explícita:

```
attributes(i)
```

```
## NULL
```

Ahora podemos usar el vector `i` para construir la matriz `m` a través de la función `matrix()`, indicando el objeto usado para crear la matriz, el número de filas y el número de columnas:

```
m <- matrix(i, nrow = 2, ncol = 3); m
```

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

Si consultamos su tipo y clase, encontramos lo siguiente:

```
typeof(m); class(m)
```

```
## [1] "integer"
```

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

Esto es, el objeto `m` es de tipo `integer`, lo que significa que está almacenado en una estructura idéntica a la usada por `i`. Pero ahora tiene una clase explícita *doble*, `matrix` y `array`. La consecuencia es que, a partir de ahora, a `m` se le aplicarán los procedimientos (métodos) diseñados para matrices y arrays, en este orden.[11]

[11] Por ejemplo, los disponibles para el álgebra matricial.

Como último ejemplo tenemos el *data frame* `df`, definido anteriormente. Ya vimos que su tipo era `list`; también dijimos que los *data frames* constituían un caso de lista *especializada*, diseñados específicamente para almacenar bases de datos. Y es razonable pensar que habrá multitud de procedimientos específicos para tratar con estos objetos, lo que se consigue gracias a los métodos definidos para su clase, `data.frame`:

```
typeof(df); class(df)
```

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

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

El tipo, no obstante, nos recuerda que todo *data frame* es, intrínsecamente, una lista, y se puede tratar (acceder, modificar, extraer...) como cualquier otra lista. Y además, se dispone de todos los procedimientos diseñados para los *data frames*.

3.3.4 *La longitud,* `length()`

La función `length()` devuelve la longitud del objeto al que se aplique. Entendemos por longitud la *cantidad de elementos del vector* considerado y, aunque la definición es unívoca, su aplicación en el caso de objetos no atómicos debe quedar clara.

En el caso de vectores atómicos es simple; el vector anterior `i` contenía los enteros del 1 al 6, luego su longitud será 6.

```
length(i)
```

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

En el caso de listas y *data frames* la longitud se refiere al número de componentes: los elementos de las listas y los elementos (o variables) de los *data frames*.

```
lst
```

```
## $A
## [1] 7
##
## $B
## [1] "uno" "dos"
##
## $C
## [1] TRUE
```

```
length(lst)
```

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

```
df
```

```
##   Id Grupo VD
## 1  1   Exp 12
## 2  2   Ctr 11
## 3  4   Ctr  9
## 4  5   Exp 14
```

```
length(df)
```

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

Por supuesto, también podemos consultar la longitud de los componentes de una lista o *data frame*.[12] Para ello aplicamos la función `length()` al elemento que deseemos:

[12] Lo que coincidiría con el número de casos.

```
length(df$Id)
```

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

Un último apunte: la función `length()`, como otras muchas en `R`, tiene un *doble uso*: sirve tanto para *consultar* la longitud de un objeto como para *establecerla*[13]. Por ejemplo, podemos reducir la longitud del vector `i` a cuatro elementos de la siguiente forma:

[13] En realidad hay *dos* funciones `length()`, pero es algo cómodamente invisible para los usuarios.

```
length(i) <- 4; i
```

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

Ejercicio 3.4.
Construye toda una serie de objetos (vectores atómicos, matriz, lista, *data frame*) y consulta para cada uno su tipo, clase y longitud.

3.3.5 *Los nombres,* `names()`

La función `names()` permite *consultar* y *establecer* los nombres de los elementos de los vectores atómicos y no atómicos. Pero lo más interesante de los nombres es que no sólo sirven para *etiquetar* los elementos, sino también para *acceder* a ellos.

Su principal utilidad se muestra al trabajar con listas y *data frames*. De hecho, ya hemos hecho uso de los nombres al construir la lista `lst` y el *data frame* `df`, y ahora podemos consultar esos nombres.

```
names(lst)
```

```
## [1] "A" "B" "C"
```

```
names(df)
```

```
## [1] "Id"    "Grupo" "VD"
```

Y podemos usarlos para acceder a su contenido. En los ejemplos anteriores, usamos el operador '`$`', como ya vimos, para enlazar el nombre del objeto con el del elemento:[14]

[14] Pensemos que en un conjunto de datos con 100 variables es más fácil recordar el nombre de una variable, por ejemplo, `datos$edad` que su índice, `datos[7]`.

```
lst$A
```

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

```
df$VD
```

```
## [1] 12 11  9 14
```

También podemos nombrar los elementos de los vectores atómicos, pero es menos común. Podemos pensar, por ejemplo, en el caso de vectores que almacenen coordenadas `(x, y)`; puede ser útil que, al mostrar su contenido, se indique a qué dimensión se refiere cada valor:

```
coord <- c(4, 12)
names(coord) <- c("x", "y")
coord
```

```
##  x  y
##  4 12
```

Los nombres pueden asignarse al crear el objeto, indicándolos sin comillas.

```
(coord <- c(x = 4, y = 12))
```

```
##  x  y
##  4 12
```

Y también podemos acceder a los elementos a través de los nombres; en tal caso hay que incluir el nombre donde habitualmente usábamos un índice numérico:

```
coord["y"]
```

```
##  y
## 12
```

3.3.6 *El modo, `mode()`*

Todos los objetos de `R` poseen un atributo, el modo, que puede obtenerse con la función `mode()`, y que en general coincide con el tipo (obtenido mediante `typeof()`).

La única diferencia en cuanto a los vectores atómicos está en los vectores de tipo `integer` y `double`, ya que ambos tienen `mode() = numeric`, por lo que puede ser de utilidad para comprobar si un vector tiene contenido numérico independientemente de su tipo.[15]

[15] Obviamente, esta comprobación deja fuera a los vectores complejos, que tienen `mode() = complex`.

```
int <- 3L
dou <- 5
typeof(int); typeof(dou)
```

```
## [1] "integer"
```

```
## [1] "double"
```

```
mode(int); mode(dou)
```

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

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

En este manual se han mantenido deliberadamente separados los atributos tipo y modo, de forma que sólo utilizaremos el tipo de los objetos. La razón es que, en la práctica, el modo es un atributo un tanto obsoleto que sólo se mantiene por compatibilidad con S.

No obstante, no es posible excluir el modo de de este manual (o cualquier otro) ya que aparece con demasiada frecuencia en manuales, webs, libros... Y en muchos casos se aprecia cierta confusión con el modo y el tipo, incluso en manuales básicos de R.[16] En el apartado 20.7 al final del libro se intentan aclarar las diferencias y similitudes.

[16] Por ejemplo, el capítulo 3 del manual *An Introduction to R* (Venables et al., 2019) utiliza ambos términos casi como sinónimos; incluso a veces de forma incorrecta.

3.3.7 Otros atributos, `attributes()`

La función `attributes()` informa de ciertas propiedades, o atributos, del objeto al que se aplica. La utilizamos en un ejemplo al estudiar la clase y en aquel caso devolvió `NULL`, ya que se trataba de un vector atómico sin ninguna característica adicional.

Pero algunos de los objetos creados en los ejemplos tienen propiedades que no hemos podido visualizar hasta ahora. Por ejemplo, la matriz `m` tiene una propiedad interesante: sus dimensiones.

```
attributes(m)
```

```
## $dim
## [1] 2 3
```

La matriz tiene dimensión `2 x 3`, como corresponde a lo indicado al construirla: `m <- matrix(i, nrow = 2, ncol = 3)`. La función `attributes()` no es la única que devuelve esta información; también podemos obtenerla con `dim()`:

```
dim(m)
```

```
## [1] 2 3
```

Los objetos `lst` y `df` también tienen atributos que mostrar. En el caso de la lista sólo se muestran los nombres de sus componentes[17].

[17] Que, como vimos, también se podían obtener con `names()`.

```
attributes(lst)
```

```
## $names
## [1] "A" "B" "C"
```

El *data frame* `df`, sin embargo, tiene más información:

```
attributes(df)
```

```
## $names
## [1] "Id"    "Grupo" "VD"
##
## $class
## [1] "data.frame"
##
## $row.names
## [1] 1 2 3 4
```

Además de los nombres (`names`, para las variables, como vimos) aparecen los nombres de filas (una secuencia que `R` añade automáticamente para cada caso) y la clase explícita.

3.3.8 *La estructura del objeto, `str()`*

Si bien las funciones anteriores informan de aspectos variados y útiles, consultar cada uno de ellos para cada objeto tratado puede ser tedioso. Una alternativa cómoda es la función `str()`, que muestra la *estructura* del objeto. La información de salida difiere según el objeto, pero en general informa de los aspectos más importantes y útiles a la hora de tratar con los objetos.

Aplicada a vectores atómicos informa de su tipo o clase, su longitud, y los primeros elementos.

```
str(1:100)
```

```
##  int [1:100] 1 2 3 4 5 6 7 8 9 10 ...
```

```
str(c(2, .3, 16))
```

```
##  num [1:3] 2 0.3 16
```

```
str(c("a", "b", "c", "d"))
```

```
##  chr [1:4] "a" "b" "c" "d"
```

A continuación aplicamos `str()` a una matriz, una lista y un *data frame*. Puede observarse cómo se muestra, para cada caso, la información más relevante.

```
str(m)
```

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

```
str(lst)
```

```
## List of 3
##  $ A: num 7
##  $ B: chr [1:2] "uno" "dos"
##  $ C: logi TRUE
```

```
str(df)
```

```
## 'data.frame':   4 obs. of  3 variables:
##  $ Id   : num  1 2 4 5
##  $ Grupo: chr  "Exp" "Ctr" "Ctr" "Exp"
##  $ VD   : num  12 11 9 14
```

Observamos que `str()` *suele* mostrar la clase, no el tipo, en su descripción del objeto, aunque no siempre, como se observa con la matriz `m`, donde aparece su tipo.[18] No obstante, viendo sus dimensiones, `[1:2, 1:3]`, se aprecia inmediatamente su naturaleza bidimensional.

[18] O su clase *implícita*. Por cierto que en la tabla 3.1 más adelante sólo se muestra la *primera* clase, `matrix`, y se ha omitido `array` por simplicidad.

Resumimos en la tabla 3.1 algunas de las propiedades vistas, aplicadas a los objetos utilizados hasta ahora. Véase que los vectores de tipo `double` tienen clase `numeric`, rompiendo con la simetría presente en el resto de vectores atómicos.

Tabla 3.1: Información sobre vectores

Objetos	typeof()	class()	str()
i	integer	integer	int [1:6] 1 2 3 4 5 6
d	double	numeric	num [1:3] 1 0.6 3
l	logical	logical	logi [1:2] TRUE FALSE
ch	character	character	chr [1:2] 'A' 'txt'
x	complex	complex	cplx [1:2] 1+2i 0+4i
m	integer	matrix	int [1:2, 1:3] 1 2 3 4 5 6
lst	list	list	List of 3 $ A: num 7 $ B: chr [1:2] 'uno' 'dos' $ C: logi TRUE
df	list	data.frame	'data.frame': 4 obs. of 3 variables: $ Id : num 1 2 4 5 $ Grupo: chr 'Exp' 'Ctr' 'Ctr' 'Exp' $ VD : num 12 11 9 14

Ejercicio 3.5.
Utiliza los objetos construidos en el ejercicio anterior y asigna nombres a los elementos de algunos de ellos. Observa el comportamiento de `names()` con matrices y *data frames*.
Luego consulta sus atributos y estructura.

3.4 Funciones genéricas

Si en el capítulo anterior presentamos las funciones, queremos ahora dedicar un breve espacio a un grupo particular de ellas: las *funciones genéricas*. La razón es que acabamos de aludir a ellas en relación al atributo `class()` y que, además, son mucho más frecuentes de lo que pudiera pensarse.

Como ya dijimos en relación a la clase de los objetos, las funciones genéricas son un tipo especial de funciones que, dependiendo de la *clase* del argumento de entrada, decide qué procedimiento (más precisamente, qué método —*method*) aplicar. Las funciones genéricas pertenecen al ámbito de la *programación orientada a objetos*, pero no es momento ahora de entrar en ningún detalle.[19]

Por ahora es suficiente saber de su existencia y objetivo, lo que ayuda a entender porqué funciones como `summary()`[20] tienen una respuesta tan diferente según el objeto. Aquí la vemos aplicada al vector lógico `l`, a la matriz `m` y a la lista `lst`.

[19] Hay todo un capítulo dedicado a la programación orientada a objetos, e inclye entre otras cuestiones cómo construir funciones genéricas y métodos asociados.
[20] Que muestra un resumen del objeto al que se aplica.

```
summary(l)
```

```
##    Mode   FALSE    TRUE
## logical       1       1
```

```
summary(m)
```

```
##        V1             V2             V3
##  Min.   :1.00   Min.   :3.00   Min.   :5.00
##  1st Qu.:1.25   1st Qu.:3.25   1st Qu.:5.25
##  Median :1.50   Median :3.50   Median :5.50
##  Mean   :1.50   Mean   :3.50   Mean   :5.50
##  3rd Qu.:1.75   3rd Qu.:3.75   3rd Qu.:5.75
##  Max.   :2.00   Max.   :4.00   Max.   :6.00
```

```
summary(lst)
```

```
##   Length Class  Mode
## A 1      -none- numeric
## B 2      -none- character
## C 1      -none- logical
```

Ejercicio 3.6.
Comprueba el funcionamiento de la función `summary()` con objetos de clase `character` y `data.frame`.

3.5 Operadores

Existen varios tipos de operadores (aritméticos, de comparación, lógicos) que iremos viendo en detalle cuando corresponda. La razón de mostrarlos ahora, además de comenzar a familiarizarnos con ellos, es atender a su *precedencia*, esto es, el orden en que se aplicarán en caso de haber varios en una misma expresión. Desatender a la precedencia lleva a no pocos errores.

Los operadores mostrados aquí están ordenados de mayor a menor precedencia. Son algunos de los que aparecen en la ayuda de `?Syntax`, elegidos por su frecuencia; el resto se verá más adelante.

Tabla 3.2: Operadores: precedencia, tipo y acción

Operador	Tipo	Número	Acción
`$`	acceso	binario	extrae componente
`[ [[`	acceso	binario	indexa vectores
`^ **`	aritmético	binario	exponenciación
`- +`	aritmético	unario	- y + unario
`:`	secuencia	binario	genera secuencia
`* /`	aritmético	binario	multiplicación y división
`+ -`	aritmético	binario	suma y resta
`< > <= >= == !=`	comparación	binario	comparaciones
`!`	lógico	unario	negación
`& &&`	lógico	binario	operador `AND`
`| ||`	lógico	binario	operador `OR`
`->`	asignación	binario	asignación hacia la derecha
`<-`	asignación	binario	asignación hacia la izquierda
`=`	asignación	binario	asignación hacia la izquierda

Olvidar las precedencias lleva a errores comunes, como el que se produce cuando, dado `n = 10`, queremos generar una secuencia desde `1` hasta `n - 1`. En una primera versión, incorrecta, la precedencia del operador `:` hace que *primero* se cree la secuencia `1:10` y *después* se reste, a cada elemento del vector, el valor `1`.

```
n <- 10
1:n - 1        # incorrecto
```

```
##  [1] 0 1 2 3 4 5 6 7 8 9
```

En la siguiente versión tomamos el control: los paréntesis fuer-

zan el cálculo de `n - 1` en primer lugar; posteriormente se crea la secuencia de `1` a `9`.

```
1:(n - 1)      # correcto
```

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

Ejercicio 3.7.

El vector `x <- runif(30)` se ha construido generando 30 valores aleatorios extraídos de una distribución uniforme entre los valores `0` y `1`. A partir de él obtén el vector `y` dados $\mu = \bar{X}$ y $\sigma = S_X$ tal que:

$$y = f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Utiliza la menor cantidad de paréntesis posible *sin* perder claridad.

3.6 Avisos y errores

Cualquier salida al ejecutar código en `R` podría catalogarse dentro de una de estas tres posibilidades:

1. Ejecución realizada y sin incidencias.
2. Ejecución realizada con un *aviso*.
3. Ejecución abortada debido a un *error*.

```
x <- c(-1, 4)
sum(x)        # Ejecutado sin incidencias
```

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

```
sqrt(x)       # Ejecutado CON aviso
```

```
## Warning in sqrt(x): Se han producido NaNs
```

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

```
x(2)          # No ejecutado POR error
```

```
## Error in x(2): no se pudo encontrar la función "x"
```

Vése que, incluso en los casos en que no hay error, es conveniente prestar atención al código[21]. La razón es que `R` es extraordinariamente robusto, lo que se traduce en que operaciones que suelen dar error en otros lenguajes, aquí son procesadas sin más. Ya mostramos un caso en que dos elementos incompatibles (de diferente *tipo*) eran agrupados en un mismo vector:

[21] Especialmente cuando estamos comenzando a trabajar con `R`.

```
v <- c("a", 4); v
```

```
## [1] "a" "4"
```

No hay aviso ni error; sencillamente, el valor `4` ha sido *coercionado* al tipo `character`, resultando en un vector de dicho tipo. Así, aunque el procedimiento se realice sin errores, puede no estar haciendo exactamente lo que queremos. Ello dependerá de que usemos los argumentos de la manera apropiada, y que estos últimos sean tratados exactamente de la forma que queremos.

También requiere atención el segundo caso: hay un aviso, pero el código se ha ejecutado y, por tanto, devuelve un resultado y el proceso continúa. Normalmente los avisos indican circunstancias particulares que pueden ser conocidas por el usuario, o no. Un ejemplo de ello es el *reciclado*[22], una operación que `R` realiza por defecto en muchos casos y que quizás no corresponde con lo que queríamos. En el caso siguiente, queremos sumar dos vectores, elemento a elemento, pero son de distinta longitud:

[22] Que veremos en detalle al final del capítulo siguiente. Básicamente, consiste en *extender* un vector reutilizando (reciclando) al final del mismo los valores iniciales hasta alcanzar una longitud determinada.

```
a1 <- c(1, 2, 3, 4)
a2 <- c(10, 20, 30)
a <- a1 + a2
```

```
## Warning in a1 + a2: longitud de objeto mayor no es múltiplo
## de la longitud de uno menor
```

Hay un aviso, cierto, pero el cálculo se ha llevado a cabo y la asignación también. Como se ve, el valor `a2[1] = 10` se ha reutilizado como sumando del cuarto elemento de `a1[4] = 4`.

```
a
```

```
## [1] 11 22 33 14
```

Este mismo ejemplo puede ilustrar la atención exigida también cuando no hay errores ni avisos. ¿Creemos que `R` nos avisará siempre que recicle algún vector?

```
a1 <- c(1, 2, 3, 4)
a2 <- c(10, 20)
a <- a1 + a2; a
```

```
## [1] 11 22 13 24
```

Parece que no. Por la salida puede inferirse que `R` sólo avisará cuando el vector mayor *no sea un múltiplo entero* del menor.

Respecto al tercer caso, los errores *abortan la ejecución*, luego atender al código y revisarlo es la única opción.

Ejercicio 3.8.
Con lo que sabemos hasta ahora, vamos a ejecutar cuatro acciones: dos de ellas con *warning* y dos que generen un error.

3.7 Más control

Para quienes entren por primera vez el mundo de la programación, esta introducción es suficiente para afrontar los siguientes capítulos con seguridad.

No obstante, aquellas personas que tengan ya experiencia en programación pueden querer hojear el capítulo 20 del mismo nombre que éste, con información que completa la que aparece aquí, como aspectos de inicio y configuración, gestión y eficiencia.

4
Vectores

Creemos haber dejado claro que los vectores[1] son el ladrillo básico de construcción en R, luego bien está dedicarle un capítulo completo para conocer en detalle sus propiedades y la forma de interactuar con ellos.

[1] Junto con las funciones, por supuesto.

RESUMEN

Creación de vectores (4.1). Funciones para su creación.
Acceso e índices (4.2). Cómo acceder a algún elemento, o a un conjunto de ellos, a partir de sus índices.
Unión de vectores (4.3). Uniones unidimensionales o bidimensionales.
Condicionales implícitos (4.4). Son pruebas lógicas aplicadas a los vectores elemento a elemento.
Creación de secuencias (4.5). Frecuentes, sirven para el control de índices en vectores hasta la creación de gráficos.
Coerción (4.6). Mecanismo por defecto en vectores atómicos, fuerza a que todos los elementos sean del mismo tipo.
Reciclado (4.7). Otro mecanismo, aplicado automáticamente cuando se realiza algún tipo de unión entre objetos de diferente longitud.

Como hemos visto en los capítulos anteriores, un vector no es más que una secuencia ordenada de elementos. Y en tanto consideremos vectores *atómicos*, sólo podrán contener elementos de la misma naturaleza (enteros, reales, caracteres, etc.)[2]

[2] Dejaremos por ahora de lado los vectores *no atómicos*, listas y *data frames*. Pero recordemos que éstos se componen de los atómicos, luego todo lo que veremos afectará a sus contenidos.

4.1 Creación de vectores

Recordemos que la forma más básica de crear un vector es combinar varios elementos mediante el operador `c()`; su tipo vendrá determinado por el tipo de los elementos que lo componen. Con la función `str()` vemos a continuación su estructura.

```
x <- c(10, 20, 30, 40, 50)
str(x)
```

```
##  num [1:5] 10 20 30 40 50
```

No obstante, la manera más habitual de obtener un vector es bien extrayéndolo de algún archivo de datos, bien como resultado de alguna función. En el ejemplo siguiente, el vector `r` contiene 4 valores aleatorios procedentes de una distribución normal estandarizada mediante la función `rnorm()`.[3]

[3] La utilización de las funciones de generación de números aleatorios es habitual en la construcción de ejemplos, por lo que la utilizaremos con frecuencia en el manual.

```
r <- rnorm(4); r
```

```
## [1] -0.9133902 -0.4517369  1.0365117 -0.2955906
```

Otra forma habitual de generar vectores es mediante funciones especialmente dedicadas, como se verá más adelante, a la creación de secuencias.

Ejercicio 4.1.
Podemos ahora generar tres números aleatorios con distribución normal con media 100 y desviación típica 15. Para ello es útil consultar la ayuda, `?rnorm`.

4.2 *Acceso e índices*

Para acceder al contenido de un vector utilizamos el operador de extracción, `[]`, introduciendo dentro de los corchetes el índice del elemento. Téngase en cuenta que en `R` los índices comienzan siempre en 1[4]:

[4] En otros lenguajes, como `C`, los índices siempre comienzan en `0`.

```
x[1]
```

```
## [1] 10
```

Podemos indicar más de un elemento; en el ejemplo usamos el operador dos puntos para extraer los elementos con índices `2`, `3`, y `4`.

```
x[2:4]
```

```
## [1] 20 30 40
```

En el siguiente caso se indican dos elementos no contiguos: `3` y `5`. Véase que el operador de extracción '`[]`', aplicado a un objeto unidimensional, espera como argumento *un único objeto*, luego los dos valores han de ser agrupados dentro de un vector mediante `c()`.

```
x[c(3, 5)]
```

```
## [1] 30 50
```

Si se quiere *acceder* a un elemento inexistente, el sistema indica que no esta accesible (`NA`); y si se hace una *asignación*, el sistema crea ese elemento y cualesquiera intermedio que sea necesario (asignándoles el valor `NA`).[5]

[5] El valor `NA` tiene un estatus especial en `R`, y es utilizado para indicar los datos perdidos (*missing data*). Es el acrónimo de *Not Available* (no disponible), y se verá en detalle en el capítulo 13.

```
x[7]            # Acceso
```

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

```
x[7] <- 70; x    # Asignación
```

```
## [1] 10 20 30 40 50 NA 70
```

Si, por el contrario, se quiere excluir algún elemento, basta indicar su índice (o índices, si son varios) precedido de un signo menos.[6]

[6] Véase que, en tanto no ha habido *asignación*, `x` permanece intacto. El único efecto es sobre el resultado mostrado en consola.

```
x[-c(3, 4)]
```

```
## [1] 10 20 50 NA 70
```

Si utilizamos los corchetes *sin indicar* ningún índice, estaremos aludiendo a *todos* los elementos de un vector.

```
x[] <- 3; x
```

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

Debe quedar clara la diferencia entre acceder a los *elementos* de un vector o acceder a éste *como un todo*. En las operaciones anteriores, se ha tratado con *uno o varios elementos* del objeto; en el siguiente caso, el *objeto en sí* es afectado hasta el punto de convertirse en otra cosa: un vector de longitud unidad que contiene el valor lógico `TRUE`.

```
x <- T; x
```

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

Ejercicio 4.2.
Definido el vector `r` a continuación, deduce la salida de las siguientes líneas de código:

```
r <- c(10, 20, 30, 40, 50, 60, 70, 80)
r[2]
r[1:5]
r[]
r[:]
r[1:]
r[:5]
r[2, 3]
r[c(2, 3)]
r[-c(2, 3)]
r[5:length(r)]
r[] <- c(2, 3); r
```

4.3 Unión de vectores

Existen diferentes formas de unir la información contenida en dos o más vectores. Pero comencemos con la versión más simple de concatenación: la función `c()`. Como ya sabemos, sirve para crear vectores indicando sus elementos. No obstante, en los ejemplos siguientes se amplían los usos que hemos visto.[7]

[7] Véase que en los ejemplo usamos paréntesis alrededor de las asignaciones; con ello se consigue una impresión del resultado sin tener que recurrir al nombre del vector recién creado.

```
(v <- c(1, 2))   # El ejemplo más simple
```

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

```
(v <- c(v, 5))   # Amplía v
```

```
## [1] 1 2 5
```

```
(w <- 10 * v)    # Creamos w a partir de v
```

```
## [1] 10 20 50
```

```
(z <- c(v, w))   # Une vectores
```

```
## [1]  1  2  5 10 20 50
```

Podemos comprobar que anidar funciones `c()` no tiene absolutamente ningún efecto[8].

[8] Habrá que esperar a estudiar las listas para construir objetos *recursivos* donde el anidamiento sí sea posible.

```
c(100, c(99, c(98, c(97))))
```

```
## [1] 100  99  98  97
```

Los ejemplos anteriores ilustran un *aumento de la longitud*, bien de un único vector, bien concatenando dos o más para crear un vector nuevo. Esto es válido para cualquier tipo de vectores atómicos, e

incluso pueden concatenarse elementos de diferente tipo, aunque en tal caso se aplicará la *coerción*[9].

Si la unión que necesitamos no requiere concatenar, sino crear una estructura bidimensional, entonces tenemos que usar una de las funciones *bind*: `rbind()` y `cbind()`, que realizan la unión por filas o por columnas, respectivamente.

[9] Que mencionamos en los capítulos anteriores y se verá en detalle en este capítulo.

```
rb <- rbind(v, w); rb
```

```
##   [,1] [,2] [,3]
## v    1    2    5
## w   10   20   50
```

```
cb <- cbind(v, w); cb
```

```
##      v  w
## [1,] 1 10
## [2,] 2 20
## [3,] 5 50
```

```
class(rb); class(cb)
```

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

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

El resultado final es bidimensional y atómico, por lo que los objetos resultantes son de clase `matrix` y `array`. Por contra, si los argumentos de `cbind()` o `rbind()` son de diferente tipo, los componentes serán coercionados. Por ejemplo, creemos el vector lógico `l`.

```
l <- c(F, F, T); l      # Vector tipo logical
```

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

Si lo unimos al vector `v`, de tipo `double`, `l` será coercionado a tipo `double`.[10]

[10] De forma que los valores `TRUE` y `FALSE` se cambian por `0` y `1`, respectivamente.

```
rb <- cbind(v, l); rb   # logical coercionado a double
```

```
##      v l
## [1,] 1 0
## [2,] 2 0
## [3,] 5 1
```

```
str(rb)
```

```
##  num [1:3, 1:2] 1 2 5 0 0 1
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:2] "v" "l"
```

Si queremos evitar este comportamiento deberemos utilizar vectores no atómicos y, por ejemplo, realizar la unión mediante `data.frame()`, de forma que cada componente mantenga su tipo original.

```
df <- data.frame(v, l); df
```

```
##   v     l
## 1 1 FALSE
## 2 2 FALSE
## 3 5  TRUE
```

```
str(df)
```

```
## 'data.frame':   3 obs. of  2 variables:
##  $ v: num  1 2 5
##  $ l: logi  FALSE FALSE TRUE
```

Existen modos más complejos de unir información, por ejemplo, cuando unimos dos *data frames* que contienen dos conjuntos de datos de un mismo estudio, pero los veremos en el capítulo 12, dedicado a este tipo de objetos.

Ejercicio 4.3.

Creamos el vector `Id` (con la identificación de cuatro pacientes: 103, 110, 156 y 212) y el vector `edad` (con las edades de los pacientes: 76, 45, 67 y 34).

Después unimos ambos de forma que tengamos una columna por variable y lo asignamos al objeto `pac`.

¿De qué *tipo* y *clase* es el objeto obtenido?

Ejercicio 4.4.

Tenemos los nombres de los cuatro pacientes anteriores (Laura, Alberto, Francisco, Berta). Podemos crear el *data frame* `pacientes` que contenga los datos de `pac` más los nombres.

¿De qué *tipo* y *clase* es el objeto obtenido?

4.4 Condicionales implícitos

Una herramienta ampliamente utilizada en `R` son los *condicionales implícitos*, que ilustramos aquí brevemente. Consisten en aprovechar la naturaleza vectorial de `R` para aplicar a un vector una condición lógica que será evaluada elemento a elemento.

```
x <- c(10, 20, 30, 40, 50)
x > 25
```

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

El resultado es un vector lógico de la misma longitud que el original donde, para cada elemento, se indica si se cumple la condición. Otra opción es solicitar los *índices* de los elementos que cumplan esa condición, lo que se consigue con la función `which()`:

```
which(x > 20)
```

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

En R suele hacerse un uso intensivo de condicionales implícitos y, bien utilizados, permiten extraer gran cantidad de información de forma muy eficiente. Aunque veremos una descripción detallada posteriormente[11], vamos a ilustrar su uso con un ejemplo muy simple. Volvamos al vector de datos `x`.

[11] En el capítulo 6, dedicado a los vectores lógicos.

```
x
```

```
## [1] 10 20 30 40 50
```

Si aplicamos una condición a `x` obtendremos un vector lógico. Pensemos en aquellos casos con valores superiores a `25`.

```
x > 25
```

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

Pues bien, ese vector lógico es el que permite operar con gran versatilidad. Por una parte sirve como índice, de forma que al usarlo sólo se seleccionarán aquellos elementos correspondientes a valores `TRUE`.

```
x[x > 25]
```

```
## [1] 30 40 50
```

Pero también podemos aprovechar los mecanismos de coerción que incorpora R. Así, al aplicar funciones matemáticas a vectores lógicos, éstos son coercionados a números[12], lo que permite contar el número de casos que cumplen la condición:

[12] A tipo `integer`, concretamente. Recordemos que los valores `TRUE` y `FALSE` se transforman en `0L` y `1L`, respectivamente.

```
sum(x > 25)
```

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

O saber su proporción:

```
mean(x > 25)
```

```
## [1] 0.6
```

Ejercicio 4.5.
Partimos del vector `edad` definido anteriormente. A partir de él podemos obtener:

- Un vector lógico donde se indique si el paciente tiene más de 50 años.
- Las edades mayores de 50 años.
- Los índices de los pacientes con más de 50 años.
- Cuántos pacientes tienen más de 50 años.

4.5 Creación de secuencias

Las secuencias tienen una presencia muy superior a lo que cabría esperar en principio. Desde operar con los índices en vectores (y matrices, *data frames*...) o gestionar los índices en los bucles, hasta crear los valores para una representación gráfica o generar patrones complejos. Y para ello contamos con tres funciones básicas como son el operador dos puntos (`:`) y las funciones `rep()` y `seq()`.

4.5.1 El operador dos puntos, :

El operador dos puntos, que vimos brevemente, genera una secuencia de números enteros entre los dos indicados, ambos incluidos, que puede ser creciente o decreciente:

```
i <- 2:6
str(i)
```

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

```
j <- 10:1; j
```

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

En un sistema vectorial, el uso de secuencias de enteros para aludir a los índices de los elementos de los vectores es muy frecuente. Así, los 50 primeros elementos del vector `datos` pueden asignarse al vector `d_primeros` simplemente escribiendo:

```
d_primeros <- datos[1:50]
```

Según los argumentos introducidos, el comportamiento de `:` puede variar, pero dejamos los detalles para el Cap. 5, donde se estudiarán en profundidad los vectores numéricos.

4.5.2 *La función `seq()`*

La función `seq()`, por su parte, crea una secuencia a partir de los argumentos de entrada. Los más básicos son:

```
seq(from = 1, to = 10, by = 2)
```

```
## [1] 1 3 5 7 9
```

Aunque su uso puede ser muy simple[13], el comportamiento de `seq()` puede llegar a ser bastante complejo, y se recomienda leer la ayuda para comprender todos los detalles. Los siguientes ejemplos muestran el uso de los argumentos de `seq()`; puede observarse que existe la costumbre, a veces, de omitir el nombre del primer argumento (o de *los primeros*, como en este caso) por ser los más comunes, pero *especificar claramente* los demás argumentos, lo que evita errores y facilita la lectura y comprensión del código en un momento posterior.

[13] Y conviene que lo sea, salvo casos excepcionales.

En su versión más simple podemos indicar simplemente el inicio y final de la secuencia, usando el valor `by = 1` por defecto. Si se indica `by` se genera la secuencia correspondiente, deteniéndose antes de alcanzar el límite si es necesario:

```
seq(from = 2, to = 14)  # De 1 en 1 por defecto
```

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

```
seq(2, 14, by = 5)      # Se detiene antes
```

```
## [1]  2  7 12
```

El argumento `along.with` hace que la secuencia tenga la misma longitud que el objeto indicado, de forma que ésta se ajustará para contener los valores necesarios para ir desde `from` hasta `to` equiespaciadamente.

```
v <- 1:5
seq(10, 13, along.with = v)
```

```
## [1] 10.00 10.75 11.50 12.25 13.00
```

El argumento `length.out` tiene un uso similar al anterior, ofreciendo una secuencia de una longitud dada. Es útil, como en el ejemplo, para generar una variable a la que se aplicará alguna función.

```
x <- seq(from = -pi, to = pi, length.out = 100)
plot(x, sin(x), "l")      # Dibuja la función
```

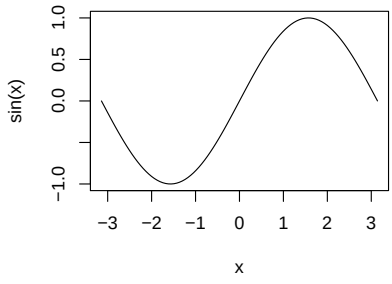

Figura 4.1: Función seno. Los valores para `x` se han obtenido como una secuencia mediante `seq()`.

Ejercicio 4.6.
Utilizaremos la función `seq()` para generar:

- v1, un vector con valores desde -3 hasta 3 a intervalos de 0.5
- v2, un vector con valores desde 3 hasta 0 a intervalos de 0.1
- v3, un vector con 10 valores equiespaciados desde 13 hasta 130
- v4, un vector con 3 valores separados por intervalos de 1/3 desde -9.67
- v5, un vector de la misma longitud que v1, con valores correlativos a partir de 1

Ejercicio 4.7.

¿Qué tipo de comportamiento tendrá la función `seq()` si se utiliza el argumento `along.with` únicamente? ¿Y si únicamente utilizamos `length.out`? Aplíquese al siguiente código.

```
seq(along.with = c(10, 20))
seq(length.out = 4)
```

4.5.3 *La función `rep()`*

La función `rep()` sirve para repetir patrones según se indique mediante sus argumentos:

```
rep(c(1, 2, 3), times = 2)   # todo, 2 veces
```

```
## [1] 1 2 3 1 2 3
```

```
rep(c(1, 2, 3), each = 2)    # cada uno, 2 veces
```

```
## [1] 1 1 2 2 3 3
```

Obsérvese que, ante la presencia de ambos argumentos, `each` necesariamente habrá de aplicarse antes que `times`.

```
rep(c(1, 2, 3), times = 2, each = 3)
```

```
##  [1] 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3
```

Puede establecerse la repetición en función de la longitud deseada, incluyendo los argumentos `times` (por defecto) o `each`.

```
rep(c(1, 2, 3), length.out = 5) # Hasta alcanzar la longitud
```

```
## [1] 1 2 3 1 2
```

```
rep(c(1, 2, 3), each = 2, length.out = 5)
```

```
## [1] 1 1 2 2 3
```

`rep()` puede utilizarse para establecer variables que indiquen, por ejemplo, niveles de factores. Supongamos que tenemos 8 casos pertenecientes a las combinaciones de 2×2 condiciones experimentales; puede construirse de la forma:

```
f1 <- rep(c(1, 2), each = 4)
f2 <- rep(c(1, 2), each = 2, times = 2)
cbind(f1, f2)
```

```
##      f1 f2
## [1,]  1  1
## [2,]  1  1
## [3,]  1  2
## [4,]  1  2
## [5,]  2  1
## [6,]  2  1
## [7,]  2  2
## [8,]  2  2
```

Ejercicio 4.8.
Ahora podemos combinar las funciones `seq()` y `rep()` para generar las siguientes secuencias:

- `5 5 5 6 6 6 7 7 7 8 8 8 9 9 9`
- `0 0 2 2 4 4 6 6 8 8 10 10`
- `5 4 3 2 1 5 4 3 2 1 5 4 3 2 1`

4.6 Objetos atómicos y coerción

`R` hace un uso frecuente de lo que se denomina *coerción*, esto es, tomar un objeto de un determinado tipo y forzar su conversión a otro tipo. El resultado de esta estrategia es doble, y conviene estar al tanto del mismo:

1. `R` es extraordinariamente robusto, en el sentido de que es capaz de *digerir* (casi) cualquier entrada mediante la coerción, sin generar errores y evitando al usuario dedicar tiempo a esas conversiones.
2. Por la misma razón, el usuario puede conocer bien esta circunstancia y aprovecharla, o no ser consciente de esas operaciones y llevarle a errores o resultados aparentemente sin sentido.

Por nuestra parte recomendamos conocer en detalle los mecanismos de coerción existentes y, además, atender las indicaciones en la ayuda de cada función, donde se indican las frecuentes coerciones realizadas.

4.6.1 Coerción implícita

Vemos a continuación las estrategias de coerción automáticas, o *implícitas*, que R lleva a cabo en los vectores que, debido a su carácter atómico, sólo permiten un tipo de contenido. Cuando mezclamos tipos de información no compatible, la estrategia general presente en R persigue no perder información, o perder la menos posible, de forma que predomina el tipo más *versátil*, esto es, aquél capaz de almacenar información más variada. En estos términos tenemos, de menor a mayor versatilidad,

```
logical < integer < double < complex < character,
```

de tal manera que al unir elementos de tipos diferentes el resultado será del tipo más a la derecha. Podemos verlo aplicado en la tabla 4.1, que muestra todas combinaciones de los cinco tipos y sus resultados.

Tabla 4.1: Coerción implícita

typeof()	logical	integer	double	complex	character
logical	TRUE	(TRUE TRUE)			
integer	2L	(2L, 1L)	(2L, 2L)		
double	3.14	(3.14, 1.00)	(3.14, 2.00)	(3.14, 3.14)	
complex	4i	(0+4i, 1+0i)	(0+4i, 2+0i)	(0+4i, 3.14+0i)	(0+4i, 0+4i)
character	'a'	('a', 'TRUE')	('a', '2')	('a', '3.14')	('a', '0+4i')	('a', 'a')

Se observa cómo el valor lógico TRUE, con la menor capacidad de almacenar información[14], es convertido en el valor 1 cuando se combina con valores numéricos, o es transformado en la cadena "TRUE" al combinarlo con un elemento alfanumérico.

En el extremo opuesto están los vectores alfanuméricos, con la máxima versatilidad[15], de forma que, cualquier elemento combinado con elementos alfanuméricos resultará automáticamente en un vector de cadena.

Todos los ejemplos anteriores de coerción implícita son realizados por R en la construcción de vectores. Pero también las funciones realizan con frecuencia algunos tipos de coerción. Lo comprobamos en el apartado sobre condicionales implícitos, donde vimos cómo algunas funciones matemáticas coercionan por defecto los valores lógicos a enteros:

```
sum(TRUE, TRUE)
```

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

Otro ejemplo puede ser el operador [], que sólo admite valores numéricos enteros, truncando los decimales si existen:

[14] En tanto sólo puede almacenar valores binarios.

[15] Algo razonable, en tanto una cadena de caracteres puede almacenar *cualquier cosa* expresable por los demás medios.

```
x <- c(10, 20, 30)
x[1.9]
```

```
## [1] 10
```

No ocurre lo mismo con los vectores alfanuméricos; no existe coerción, y las comparaciones pueden ser engañosas.

```
"3" + "4"
```

```
## Error in "3" + "4": argumento no-numérico para operador binario
```

```
"3" < "4"
```

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

```
"33" < "4"
```

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

Se observa que la comparación actúa sobre las *cadenas* `"33"` y `"4"`, y no sobre los *números* `33` y `4`.

4.6.2 Coerción explícita

Frente a la coerción implícita tenemos la *coerción explícita*, que impone un tipo particular al objeto indicado. Se lleva a cabo con la familia de funciones `as.xxx`, donde `xxx` puede tomar los valores `integer`, `double`, `logical`, `character`, `complex`, etc. (véase la ayuda para conocer todas las opciones)[16]. En el capítulo dedicado a manipulaciones veremos un apartado (17.5) dedicado a esta amplia familia de funciones.

La coerción explícita se utiliza con frecuencia en funciones, para garantizar que el argumento introducido es del tipo adecuado. Podemos verlo aplicado en la tabla 4.2.

[16] Existe otra familia de funciones similar, `is.xxx`, cuyo objetivo es evaluar si un objeto es de un determinado tipo. El resultado es un valor lógico indicando si se cumple la condición. Las iremos viendo a cuando tratemos los diferentes objetos.

Tabla 4.2: Coerción explícita

typeof()		as.logical	as.integer	as.double	as.complex	as.character
logical	TRUE		1	1	1+0i	'TRUE'
logical	FALSE		0	0	0+0i	'FALSE'
integer	2L	TRUE		2	2+0i	'2'
double	3.14	TRUE	3		3.14+0i	'3.14'
complex	2.1+4i	TRUE	2	2.1		'2.1+4i'
character	'a'	NA	NA	NA	NA	
character	'T'	TRUE	NA	NA	NA	
character	'2.1'	NA	2	2.1	2.1+0i	

Se observa cómo el comportamiento de estas funciones frente a los vectores alfanuméricos es más complejo, ya que las funciones *comprueban* si el contenido de las cadenas de caracteres es asimilable a su tipo. Así, `as.logical()` reconoce las cadenas `'T'` y `'TRUE'`, mientras que las funciones numéricas reconocen e interpretan las cadenas con contenido numérico.

Ejercicio 4.9.
A continuación se presenta una serie de definiciones y acciones. El objetivo es predecir el resultado a obtener al ejecutar cada línea de código, indicando en especial tipo y contenido del resultado obtenido.

```
vec <- seq(2, 12, length.out = 6)
vec <- as.double(vec)
vec[5] <- "a"
vec <- as.integer(vec)
vec[5] <- T
vec <- as.double(vec)
vec <- vec/2
as.logical(vec)
as.logical(as.integer(vec))
```

Ejercicio 4.10.
Si tenemos la cadena de caracteres `ch <- "1.7+3i"`. ¿Qué resultado podemos esperar obtener de aplicarle las siguientes funciones: `as.integer()`, `as.double()` y `as.complex()`?

4.7 *Reciclado*

El *reciclado* es un procedimiento por el que el sistema extiende un vector hasta alcanzar la longitud de otro, añadiendo elementos de forma circular o cíclica.

El reciclado es usado por defecto en muchas operaciones que requieren más de un vector. Por ejemplo, como ya vimos, la suma de vectores es una operación que se realiza elemento a elemento:

```
v1 <- c(1, 2)
v2 <- c(10, 20)
v1 + v2
```

```
## [1] 11 22
```

Pero, ¿qué ocurre cuando los vectores tienen distinta longitud?

```
x <- c(1, 2, 3)
y <- c(10, 20)
z <- 100
x + y + z
```

```
## Warning in x + y: longitud de objeto mayor no es múltiplo de
## la longitud de uno menor
```

```
## [1] 111 122 113
```

Si se observa, el resultado es equivalente a haber sumado los vectores del siguiente ejemplo. Véase que `yy` y `zz` no son más que `y` y `z`, a los que se han añadido sus propios elementos (*reciclándolos*) hasta alcanzar una longitud igual a 3.

```
x  <- c(  1,   2,   3)
yy <- c( 10,  20,  10)
zz <- c(100, 100, 100)
x + yy + zz
```

```
## [1] 111 122 113
```

Como vimos en el apartado referido a avisos y errores, el sistema da una aviso (*Warning*), pero ejecuta la acción. Al igual que en aquella ocasión, no siempre da un aviso; sólo cuando los vectores no son múltiplos enteros.

```
X <- 1:8
Y <- seq(10, 40, by = 10)
Z <- c(100, 200)
X; Y; Z
```

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

```
## [1] 10 20 30 40
```

```
## [1] 100 200
```

```
X + Y + Z
```

```
## [1] 111 222 133 244 115 226 137 248
```

El reciclado no sólo tiene lugar al operar sobre varios vectores; también ocurre en la construcción de nuevos objetos a partir de vectores de diferente longitud, como es el caso con objetos de clase `matrix` o `data.frame`. Al construir una matriz mediante `cbind()`, el reciclado lleva a buen término la tarea, aunque da un aviso cuando los vectores no tienen múltiplos enteros entre sí.

```
cb <- cbind(x, y, z); cb
```

```
## Warning in cbind(x, y, z): number of rows of result is not a
## multiple of vector length (arg 2)
```

```
##      x  y   z
## [1,] 1 10 100
## [2,] 2 20 100
## [3,] 3 10 100
```

```
CB <- cbind(X, Y, Z); CB
```

```
##      X  Y   Z
## [1,] 1 10 100
## [2,] 2 20 200
## [3,] 3 30 100
## [4,] 4 40 200
## [5,] 5 10 100
## [6,] 6 20 200
## [7,] 7 30 100
## [8,] 8 40 200
```

Al construir un *data frame*, sin embargo, no se exige igualdad de longitud, pero sí que sean múltiplos enteros. En caso contrario, la ejecución da error, y no se lleva a cabo.

```
df <- data.frame(x, y, z)
```

```
## Error in data.frame(x, y, z) :
```

```
##   arguments imply differing number of rows: 3, 2, 1
```

```
DF <- data.frame(X, Y, Z); DF
```

```
##   X  Y   Z
## 1 1 10 100
## 2 2 20 200
## 3 3 30 100
## 4 4 40 200
## 5 5 10 100
## 6 6 20 200
## 7 7 30 100
## 8 8 40 200
```

Ejercicio 4.11.
Al igual que en ejercicios anteriores, se presentan una serie de definiciones y acciones. El objetivo es predecir el resultado a ob-

tener, atendiendo especialmente a si las uniones son factibles o no y, en caso afirmativo, qué tipo de resultado se obtiene.

```
dos <- c(100, 200); dos
tres <- c(10, 20, 30); tres
cuatro   <- c(1, 2, 3, 4); cuatro
dos + tres
cbind(dos, tres)
data.frame(dos, tres)
cbind(dos, cuatro)
data.frame(dos, cuatro)
cbind(tres, cuatro)
data.frame(tres, cuatro)
cbind(dos, tres, cuatro)
data.frame(dos, tres, cuatro)
```

5
Vectores para información numérica

La información numérica puede ser almacenada en `R` en tres formatos básicos: de tipo entero (`integer`), reales (tipo `double`, o valores de coma flotante) y números complejos (`complex`). En este capítulo nos centraremos en los *vectores atómicos* de estos tipos, pero sus características y comportamiento serán iguales cuando estos vectores estén configurados como matrices o *arrays* o cuando formen parte de listas o *data frames*.

Complementamos lo anterior con cuestiones básicas (como las operaciones numéricas y el tratamiento de los decimales) y otras avanzadas (como operar con números expresados en base no decimal, o aspectos de precisión numérica).

RESUMEN

Vectores enteros (5.1), reales (5.2) y complejos (5.3). Creación y características de cada uno.

Operaciones con vectores numéricos (5.4). Veremos operadores aritméticos y relacionales, y funciones matemáticas.

Tratamiento de los decimales (5.5). En cuatro vertientes: el tratamiento del carácter *coma* decimal, el formato visual, la precisión del valor almacenado y las comparaciones.

Codificación binaria, hexadecimal y octal (5.6). Formatos de codificación numérica.

Precisión numérica (5.7). Sobre los límites de toda codificación y sus efectos.

Generalmente, `R` gestionará las variables numéricas habituales (`double` e `integer`) que utilicemos de forma apropiada y pocas veces necesitaremos conocer o cambiar su clase. Por defecto, `R` utiliza el tipo `double` siempre que definamos un valor numérico, aunque no tenga decimales:

```
n <- 7
typeof(n)
```

```
## [1] "double"
```

Si queremos definir valores de tipo `integer` deberemos hacerlo explícitamente, o usando las funciones apropiadas, como vemos a continuación.

5.1 Vectores enteros

Existen al menos cuatro maneras de crear un vector de tipo `integer` que contenga, por ejemplo, el valor 7. El resultado de todas ellas es el mismo, aunque los modos de operar sean muy distintos:

```
i1 <- 7L              # sufijo L
i2 <- 7:7             # operador dos puntos
i3 <- seq(7, 7)       # función seq()
i4 <- as.integer(7)   # coerción a integer
```

La primera forma impone el tipo `integer` mediante el sufijo `L`[1], y es la más directa de las cuatro. Las dos siguientes usan funciones para crear secuencias que, por defecto, devuelven valores enteros: el operador dos puntos, siempre; la función `seq()`, cuando los valores inicial, final e intermedios *no tengan decimales* (si los tienen se almacenará con tipo `double`). En ambos casos, su uso es un tanto artificial (una secuencia *desde 7 hasta 7*) pero cumple el objetivo. La función `as.integer()`, por último, toma un valor por defecto real (7) y lo coerciona a `integer`.

[1] El motivo para usar el carácter `L` como indicador de vectores de tipo `integer`, parece estar en que, al inicio de la construcción de `R`, los enteros estaban codificados por defecto en formato *largo* (*Long*; con 32 bits en lugar de los habituales 16 bits por defecto en aquella época). Véase al respecto http://stackoverflow.com/questions/22191324/clarification-of-l-in-r/22192378#22192378.

La función `integer()`, por su parte, tiene como función crear un vector de la longitud indicada como argumento, y no hay que confundirla con `as.integer()`. La primera tiene utilidad para crear vectores de cierta longitud (con todos sus valores a cero) generalmente para asignarle sus valores posteriormente.

```
i5 <- integer(7); i5
```

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

Véase a continuación que el sufijo `L` no permite coaccionar a `integer` un número con decimales significativos. La función `as.integer()`, por el contrario, sí consigue la coerción.

```
str(7.0L)
```

```
##  int 7
```

```
str(7.7L)
```

```
##  num 7.7
```

```
str(as.integer(7.7))
```

```
##  int 7
```

Los enteros se representan en R mediante 32 bits, luego pueden tomar 2^31 valores posibles en valor absoluto, ya que un bit se reserva para el signo. Por tanto, si consideramos valores negativos y positivos estarán entre -(2^31) + 1 y (2^31) - 1, esto es, en el rango de -2147483647 a 2147483647. La variable del sistema .Machine$integer.max nos informa del máximo entero posible:[2]

[2] Para obtener el mínimo basta cambiar de signo.

```
(max_int <- .Machine$integer.max)
```

```
## [1] 2147483647
```

Si excedemos esos límites, el valor no podrá almacenarse en formato integer; el sistema da un aviso y devuelve NA.

```
max_int + 1L
```

```
## Warning in max_int + 1L: NAs producidos por enteros
## excedidos
```

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

Una opción es, en caso de necesidad, pasar al formato double; en el ejemplo anterior, basta que el valor sumado no tenga el sufijo L.

```
str(max_int + 1)
```

```
##  num 2.15e+09
```

Otra opción, si por alguna razón hubiera que exceder ese límite manteniendo el tipo integer, sería acudir al paquete bit64 (Oehlschlägel, 2017)[3], como se observa a continuación:

[3] https://CRAN.R-project.org/package=bit64.

```
install.packages("bit64")
library(bit64)
(i64 <- as.integer64(.Machine$integer.max) + 1L)
```

```
## [1] 2147483648
```

```
str(i64)
```

```
## integer64 2147483648
```

Ejercicio 5.1.
A continuación se muestran varios vectores. Convierte a un vector entero aquello que *no sean* ya enteros. Observa cuáles pueden coercionarse y cuáles no, y los resultados.

```
v1 <- c("1", "3.14")
v2 <- 1:12
v3 <- seq(2, 6)
v4 <- seq(0, 5, length.out = 5)
v5 <- rnorm(10)
m  <- matrix(1:12, 3, 4)
```

5.2 Vectores reales

El tipo `double`, clase `numeric`, almacena lo que entendemos como números reales[4], codificados en formato de *coma flotante*[5], y constituye el tipo de objeto por defecto para almacenar valores numéricos.

Si queremos coercionar algún vector (generalmente de enteros) a dobles podemos usar la función `as.double()`.

```
int <- 1:6
(dou <- as.double(int))
```

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

La salida, como se observa, es indiferenciable de la ofrecida por el vector `int` ya que ningún elemento posee decimales. Para confirmar el cambio debemos comprobar su tipo, clase o usar `str()`.

```
typeof(dou); class(dou)
```

```
## [1] "double"
```

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

```
str(dou)
```

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

Vemos que, a diferencia del resto de vectores atómicos, el tipo y la clase no coinciden (`double` y `mumeric`) y la función `str()` muestra `num` y no `dou`, por ejemplo.

Las variable de tipo doble se codifican mediante 64 bits. Los valores más grandes y más pequeños representables mediante este tipo de vectores pueden obtenerse, como antes, mediante la variable de sistema `.Machine` (téngase en cuenta que los valores exactos pueden variar según la máquina).

[4] Al igual que en caso de los enteros y el sufijo `L`, la denominación de *double* procede de los bits utilizados por defecto para su codificación (64 bits), lo que constituía el *doble* de la codificación habitual (o *single*, de 32 bits). Al contrario del sufijo `L` para indicar enteros, esta nomenclatura es genérica en el ámbito informático, y no específica de `R`.

[5] La notación de coma flotante permite expresar número extremadamente grandes y pequeños, y está basada en la notación científica. Véase https://es.wikipedia.org/wiki/Coma_flotante

```
.Machine$double.xmin
```

```
## [1] 2.225074e-308
```

```
.Machine$double.xmax
```

```
## [1] 1.797693e+308
```

En el caso de los vectores enteros, vimos que sumar una unidad (explícitamente `integer`) al valor máximo posible llevaba a la pérdida del valor, de modo que se asignaba `NA`. Con los vectores de contenido `numeric`, sin embargo, exceder el valor máximo lleva a otro tipo de resultado[6]:

```
str(.Machine$double.xmax * 10)
```

```
##  num Inf
```

Es importante notar que el valor `Inf` obtenido es plenamente funcional: es un número, almacenado como `double` y con el que se puede operar, aunque con todas sus particularidades. Lo mismo ocurre con `-Inf`.

```
Inf + 100
```

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

También podemos usar la notación científica para expresar cualquier numero. El resultado es un vector de tipo `double` sin mayores atributos; por tanto, esta notación es simplemente una forma de expresar cantidades numéricas, y no constituye un tipo particular de codificación en R[7].

```
e <- 2.45e-2
str(e)
```

```
##  num 0.0245
```

Ejercicio 5.2.
Una de las cinco constantes que implementa `R` es `pi`. Y uno de los usos posibles es crear una secuencia de valores entre -2π y 2π para hacer un gráfico, por ejemplo, de la función coseno.

5.3 *Vectores complejos*

Si no necesitas utilizar números complejos, o incluso si no has oído hablar de ellos, nuestra recomendación es saltar este apartado. En

[6] ¿Por qué no hemos utilizado el mismo prodimiento anterior, sumando un `1` al valor máximo? Porque el procedimiento utilizado por `R` (o por cualquier sistema informático) en tales casos permitiría la codificación, aparente, pero sin que hubiera una *verdadera codificación, exacta* de esa unidad añadida. Puede entenderse por completo si leemos, más adelante, el apartado 5.7

[7] En algunos lenguajes se permite usar, además de la `e`, el carácter `d`, pero *no* en `R`

```
e <- 2.45d-2
## Error: unexpected symbol in
## 'e <- 2.45d'
```

caso contrario, estamos de suerte, ya que R da soporte a los *números complejos* de manera cómoda.

Comencemos recordando que un número complejo[8] es un número con dos componentes (una parte *real* y otra *imaginaria*) que puede representarse de la forma $x + yi$ donde x e y son números reales e i es el valor que satisface la igualdad $i^2 = -1$.

Un número complejo puede representarse como un punto de coordenadas (x, y) en el plano complejo, lo que se conoce como *diagrama de Argand*. El primer elemento representa la parte real (x) y el segundo la parte imaginaria (yi)[9].

La expresión anterior, $z = x + yi$, se conoce como *forma algebraica*, cartesiana o rectangular. Otra forma de representar un número complejo es mediante la *forma polar*, cuyos componentes son el *módulo* (r) y el *argumento* (φ), de modo que $z = r(\cos\varphi + i \sin\varphi)$. La figura 5.1 muestra los dos tipos de componentes del número complejo z.

Existen varias formas de definir un número complejo en R. La más simple es, como ya hemos usado en ocasiones, la forma algebraica. R reconoce el sufijo `i` junto a un valor numérico (sin espacios) y crea un objeto de tipo `complex`[10]:

```
z <- 4+3i
str(z)
```

```
##  cplx 4+3i
```

Otra forma es mediante la función `complex()`, que permite especificar un número complejo tanto en forma cartesiana como polar:

```
z1 <- complex(real = 4, imaginary = 3)
z2 <- complex(modulus = 2, argument = pi/4)
```

La forma polar también permite expresar z usando la fórmula de Euler: $z = re^{i\varphi}$. Si $r = 2$ y $\varphi = \pi/4$, entonces:

```
z2 <- 2 * exp(1i * pi/4)
```

De igual forma que creamos un número complejo, podemos obtener sus componentes en ambos formatos: algebraico y polar.[11] . Las funciones para ello son:

```
Re(z1); Im(z1)
```

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

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

```
Mod(z1); Arg(z1)
```

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

[8] Véase https://en.wikipedia.org/wiki/Complex_number.

[9] A veces se utiliza el carácter j para indicar la parte imaginaria del complejo, especialmente en algunas áreas de la ingeniería. En R sólo puede utilizarse i.

[10] Véase que R omite por defecto los espacios alrededor del operador suma (+) cuando se trata de expresar complejos.

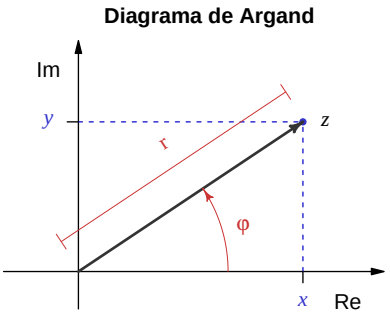

Figura 5.1: Componente de las formas cartesiana (azul) y polar (rojo) al representar el complejo z. El eje de abcisas representa el componente real y, el de ordenadas, el componente imaginario.

[11] El *módulo*, r, indica la longitud del vector z, y se obtiene mediante la expresión $r = |z| = \sqrt{x^2 + y^2}$. El *argumento*, φ, corresponde al ángulo que forma el vector z (en *radianes*) y cuyo cálculo a partir de x e y depende del cuadrante donde se sitúe z (Véase su cálculo en https://en.wikipedia.org/wiki/Complex_number).

```
## [1] 0.6435011
```

Por último, el *conjugado complejo* de z se nombra $\bar{z}$ y se define como $\bar{z} = \text{Re}(z) - \text{Im}(z)\ i$. En R podemos obtenerlo mediante la función `Conj()`.

```
Conj(z1)
```

```
## [1] 4-3i
```

Como con otros tipos de objetos, R dispone de las funciones `as.complex()` e `is.complex()`, que sirven, respectivamente, para coercionar a tipo `complex` y comprobar si un objeto pertenece a este tipo.

Véase en el ejemplo que la función `as.complex()` hace que el objeto al que se aplica sea asignado a dicho tipo aunque, obviamente, *no posee parte imaginaria*, por lo que ésta será igual a `0i`.

```
(vx <- as.complex(-9))
```

```
## [1] -9+0i
```

```
is.complex(vx)
```

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

Su utilidad más inmediata es obtener soluciones cuando el espacio de los números reales no contiene la solución requerida:

```
sqrt(as.complex(-9))
```

```
## [1] 0+3i
```

Al final del capítulo 5 del manual de *Ejercicios resueltos y comentados* hay un ejercicio complementario para poner en práctica el uso de vectores complejos en R.

5.4 Operaciones con vectores numéricos

Por defecto, las operaciones aritméticas que incluyen dos (o más) vectores numéricos se realizan *elemento a elemento*, como hemos podido comprobar:[12]

[12] Recordemos que, si las longitudes no son iguales, se aplicará el *reciclado* por defecto.

```
v1 <- 1:5
v2 <- seq(10, 50, by = 10)
v1 + v2
```

```
## [1] 11 22 33 44 55
```

A continuación se muestran algunos operadores (aritméticos y relacionales) y funciones matemáticas.

5.4.1 Operadores aritméticos

Los operadores aritméticos (unarios y binarios) son:

Tabla 5.1: Operadores aritméticos

	Operación		Operación		Operación
`+x`	Positivo	`x - y`	Resta	`x ^ y`	Exponenciación
`-x`	Negativo	`x * y`	Producto	`x%% y`	x mod y
`x + y`	Suma	`x / y`	División	`x%/% y`	División entera

La mayoría de los operadores ya se han utilizado y son de uso común, así que mostramos las operaciones menos conocidas. Véase que la exponenciación (`^`) también puede indicarse mediante el operador (`**`). El operador `%/%` ofrece la parte entera del cociente resultante, mientras que `%%` nos da el resto de la división entera.

```
2 ** 8      # Exponenciación
```

```
## [1] 256
```

```
19/7        # División
```

```
## [1] 2.714286
```

```
19 %/% 7    # División entera
```

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

```
19 %% 7     # Resto de la división entera
```

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

Véase que estos operadores (excepto `'%%'` y `'%/%'`) son utilizables igualmente con números complejos, donde se aplica la aritmética compleja por defecto.

```
z1 <- 3+2i
z2 <- 1-1i
z1 + z2     # Suma
```

```
## [1] 4+1i
```

```
z1 * z2     # Producto
```

```
## [1] 5-1i
```

```
z1 / z2     # División
```

```
## [1] 0.5+2.5i
```

```
z1 ^ z2     # Exponenciación
```

```
## [1] 4.987933-4.154361i
```

Ejercicio 5.3.
Dado el vector `v <- 1:12`, sustituye por cero los múltiplos de 3.

5.4.2 *Funciones matemáticas básicas*

R posee una ingente cantidad de funciones, tanto en la distribución base como en los paquetes disponibles, por lo que un listado completo puede ser poco práctico. Las que siguen son algunas funciones comunes, elegidas sólo con afán ilustrativo[13]. Por claridad, hemos omitido los paréntesis.

[13] Muchas de ellas aparecen en la ayuda bajo el título de *S3 Group Generic Functions*. Para acceder, podemos teclear en la ventana de ayuda `S3groupGeneric`.

Tabla 5.2: Algunas funciones matemáticas

abs	floor	sqrt	sin	sum	cumsum	mean
sign	ceiling	exp	cos	prod	cumprod	median
	trunc	log	tan	max	cummax	sd
	round	log2	asin	min	cummin	var
		log10	acos	range		cor
			atan			

Iremos viendo algunas de ellas más adelante. Una es `round()`, en la columna de las funciones dedicadas a redondeo, que permite indicarlos decimales a mantener:

```
round(54321.12345, 2)
```

```
## [1] 54321.12
```

Cuando se requieran funciones más especializadas podremos buscarlas bien mediante `?función` o `??función`, bien a través de algún buscador. En la ayuda podremos encontrar información sobre los argumentos y los resultados que proporciona.

Ejercicio 5.4.
Calcula el sumatorio y el productorio del vector `v <- 1:4`.

5.4.3 Operadores relacionales

Los operadores relacionales también actúan elemento a elemento, y ofrecen resultados de tipo `logical` (véase la tabla 5.3). Ponen a prueba relaciones del tipo mayor/menor e igualdad/desigualdad y son, entre otras utilidades, la base de los condicionales implícitos vistos anteriormente.

Tabla 5.3: Operadores relacionales

	Prueba		Prueba
`<`	Menor	`==`	Igualdad
`<=`	Menor o igual	`!=`	Desigualdad
`>`	Mayor	`!x`	No `x`
`>=`	Mayor o igual		

Veamos algunos ejemplos:

```
x <- 0:5; x
```

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

```
x < 3           # Menor que
```

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

```
x >= 3          # Mayor o igual que
```

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

```
y <- c(1, 1, 2, 4, 4, 4)
x != y          # Distinto
```

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

```
x[x < 3]
```

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

Se observa cómo, salvo la última línea de código, la salida siempre es de tipo `logical`. En la última línea se utiliza un condicional implícito (`x < 3`) para obtener los índices de los elementos mostrados.

El operador *negación*, `!` (o $\neg x$), es un caso especial en tanto es unario (esto es, no pone *dos* objetos en relación). Se incluye aquí por razones de consistencia, pero hay que tener presente su funcionamiento: invierte los valores lógicos:

```
!c(T, F, T)
```

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

Existen también los *operadores lógicos* pero, en tanto operan con vectores de tipo `logical` (o coercionan a `logical` el argumento de entrada), se verán en el próximo capítulo.

Ejercicio 5.5.
Obtenemos los vectores `v1` y `v2` muestreando 5 elementos del vector `v <- 0:10` mediante la instrucción `sample()`. Indica, para cada elemento de `v1`, si es mayor que el elemento correspondiente de `v2`.

5.4.4 Algebra matricial

Dejaremos las operaciones matriciales para cuando conozcamos mejor estos objetos: el capítulo 9, dedicado a matrices y *arrays*.

5.5 Tratamiento de los decimales

Como vimos en relación a los vectores numéricos, los valores reales utilizan una codificación de coma flotante, y sus detalles, incluidos los valores máximos y mínimos, ya se estudiaron entonces[14]. No obstante, existen cuatro aspectos a tener en cuenta en cuanto al trabajo con los decimales y que se comentarán por separado.

[14] Y se ampliará posteriormente en la sección 5.7 sobre la precisión numérica.

El primero de ellos es anecdótico (aunque de conocimiento obligado) y se refiere al uso del carácter coma decimal en algunos entornos, frente al punto decimal utilizado en `R` por defecto. Aunque es una cuestión de formato, suele generar errores que hay que conocer.

El segundo aspecto se refiere a cuestiones de redondeo, pero a efectos únicamente de *formato*, esto es, el número de decimales mostrado (con independencia de los decimales codificados), mientras que el tercero trata del *redondeo* a efectos de codificación del número, lo que implica eliminación de decimales (con el consiguiente cambio en la precisión de los valores utilizados), algo que requiere mayor cuidado.

Por último, veremos la forma adecuada de hacer comparaciones para que no nos afecten cuestiones derivadas de la codificación.

5.5.1 La coma decimal

Frente al estándar del punto decimal, operar con la coma decimal es, más que otra cosa, un pequeño engorro que requiere atención adicional. Por ejemplo, es frecuente importar datos que, al contener comas decimales, son interpretados como texto y automáticamente codificados como tipo `character`[15].

[15] Y, en el caso de importar a un *data frame*, es posible que acaben convertidos en objetos de clase `factor`.

La forma de trabajar en estos casos es usar un argumento que indique este hecho, por ejemplo mediante el argumento `dec` en funciones de lectura y escritura de datos:[16]

[16] Que veremos en su momento. Por ahora baste conocer este detalle.

```
read.table("Datos.txt", dec = ",")
```

También es posible, de modo general, cambiar el modo en que `R` genera las salidas a través del argumento `OutDec` en la función `options()`[17], pero ello no afectará al tratamiento de los valores decimales. Es decir, aunque *muestre* la coma decimal sigue *operando* únicamente con el punto decimal.

[17] Puede ser interesante echar un vistazo a las opciones disponibles, bien para modificarlas, bien simplemente para conocer su existencia y sus valores por defecto. La manera más fácil de ver las opciones junto con una descripción de cada una es usar la ayuda:
`?getOption`

```
options(OutDec= ",")
print(2.3)
```

```
## [1] 2,3
```

```
2,3 + 1
```

```
## Error: <text>:1:2: inesperado ','
## 1: 2,
##      ^
```

Esta opción es útil, por ejemplo, para publicaciones que exijan el formato de coma decimal. Posteriormente, es conveniente devolver la opción `OutDec` a su estado original[18].

[18] En caso de no hacerlo, `R` volverá al estado por defecto al reiniciarse.

5.5.2 *El formato visual de los decimales*

Por defecto, el número de dígitos que muestra `R` es de `7`. Como puede comprobarse, este límite se aplica a la cantidad de dígitos *significativos* o relevantes con cierta flexibilidad.

```
1/3
```

```
## [1] 0.3333333
```

```
1000/3
```

```
## [1] 333.3333
```

Como otras tantas cuestiones, este comportamiento viene definido por un parámetro que podemos obtener de la siguiente forma:

```
getOption("digits")
```

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

Si modificamos el parámetro, la salida cambia aunque, como se indica en la ayuda, el valor indicado será sólo una *sugerencia*, esto es, que el número de caracteres mostrados puede variar en función de otros criterios (generalmente evitar la pérdida de información relevante). Esto puede observarse en el ejemplo siguiente:

```
options(digits = 2)
a <- 1/3; a
```

```
## [1] 0.33
```

```
b <- 1000/3; b
```

```
## [1] 333
```

Esta función sólo afecta al número de dígitos *mostrados*; como es de esperar, el número de dígitos *almacenados* mantiene su precisión original[19]:

```
options(digits = 12)
a; b
```

```
## [1] 0.333333333333
```

```
## [1] 333.333333333
```

Una de las utilidades básicas del control de decimales es, por ejemplo, presentar los resultados de acuerdo con los estándares de publicación. Un caso frecuente sería la presentación de una matriz de correlaciones:

```
options(digits = 7)  # Por defecto
a1 <- rnorm(5); a2 <- rnorm(5); a3 <- rnorm(5)
a  <- cbind(a1, a2, a3)
cor(a)
```

```
##           a1        a2        a3
## a1 1.0000000 0.8638040 0.3513575
## a2 0.8638040 1.0000000 0.4016519
## a3 0.3513575 0.4016519 1.0000000
```

```
options(digits = 3)
cor(a)
```

```
##       a1    a2    a3
## a1 1.000 0.864 0.351
## a2 0.864 1.000 0.402
## a3 0.351 0.402 1.000
```

[19] El parámetro `digits` acepta valores entre 1 y 22. Forzar a valores máximos, de hecho, puede llevarnos a algunas sorpresas, por ejemplo:

```
options(digits = 22)
a
## [1] 0.33333333333333331483
```

Las razones de este extraño resultado pueden consultarse en el último apartado de este capítulo, aunque queda un poco fuera del alcance de este tema en una primera lectura.