

Taller de Introducción a R

Felipe Ortega María Jesús Algar
Emilio López Cano

2024-10-03

Tabla de contenidos

Introducción	1
Licencia	2
 I. Instalación y conceptos básicos	 3
1. El lenguaje R	5
1.1. ¿Qué es R?	5
1.2. RStudio	5
1.3. ¿Cómo se mantiene R en la actualidad?	6
1.4. Ventajas e inconvenientes de R.	6
1.5. Paquetes R	7
1.6. Fuentes de información sobre R	9
 2. Instalación de R y R Studio	 11
2.1. Trabajando con RStudio	11
2.2. Proyectos de RStudio y directorio de trabajo	18
 3. Organización de proyectos	 25
3.1. Estructura de directorios	25
3.2. Tipos de proyectos	27

II. El lenguaje R	29
4. Programación básica en R	31
4.1. Objetos y variables.	31
4.1.1. Función <code>ls()</code>	33
4.1.2. Función <code>rm()</code>	33
4.1.3. Operadores aritméticos.	33
4.1.4. Operadores lógicos.	33
4.2. Funciones.	33
4.2.1. Parámetros con valores por defecto.	35
4.2.2. Llamada a una función.	36
5. Tipos de datos	39
5.1. Vectores.	39
5.1.1. Creación de vectores.	39
5.1.2. Valores faltantes.	43
5.1.3. Indexación de elementos	44
5.1.4. Vectorización.	48
5.2. Factores.	49
5.3. Matrices y Arrays.	52
5.3.1. Construcción de arrays a partir de vectores.	52
5.3.2. Función <code>matrix()</code>	53
5.3.3. Función <code>array()</code>	53
5.3.4. Nombres de filas y columnas.	54
5.3.5. Indexación de elementos.	55
5.3.6. Unión de matrices y vectores	57
5.3.7. Operaciones aritméticas	58
5.4. Listas.	60
5.4.1. Creación de una lista.	60
5.4.2. Indexación de elementos.	61
5.4.3. Añadir y quitar elementos a una lista.	63
5.5. Data Frames.	65
5.5.1. Creación de un Data Frame.	66
5.5.2. Características de un Data Frame.	67

5.5.3.	Indexación de elementos.	68
5.5.4.	Añadir nuevas columnas.	70
5.5.5.	Extensión de Data Frames: tibble.	71
6.	Tidy data	79
6.1.	Principios: “tidy data”	80
6.2.	El paquete tidyverse	83
6.2.1.	Resumen de paquetes principales en tidyverse	84
6.2.2.	Método de trabajo con tidyverse	85
6.2.3.	Objetos tibble	86
6.3.	Preparación de datos con tidyverse: tidyr	88
6.4.	Consulta, manipulación y actualización de datos: dplyr . . .	92
6.4.1.	Funciones principales en dplyr	93
6.4.2.	Ejemplos de uso de dplyr	94
7.	Importación y exportación de datos	111
7.1.	Importación de datos	111
7.1.1.	Importar datos de archivos de texto: CSV	112
7.1.2.	Importar datos de archivos Excel	117
7.1.3.	Otras herramientas de importación de datos	117
III.	Ejemplo práctico	119
8.	Análisis exploratorio de datos	121
8.1.	Datos de ejemplo	121
8.2.	Carga de datos y esquema (tipos de datos)	122
8.3.	Operaciones básicas	125
8.3.1.	Operaciones de selección	125
8.3.2.	Transformación de datos	130
8.3.3.	Combinación de tablas de datos (funciones join) . .	131
8.4.	Resúmenes estadísticos	131
8.5.	Gráficos exploratorios	131

Tabla de contenidos

9. Creación de gráficos	133
10. Modelos con R	135
11. Recursos adicionales	137
11.0.1. Ejercicios y referencias sobre <code>tidyverse</code>	137
11.1. Recursos para investigación en Agricultura	138
Referencias	139
 Apéndices	 141
A. Comandos de utilidad	141
A.1. Comandos en R Studio	141
A.2. Comandos con <code>dplyr</code>	141
B. Entornos de desarrollo	143
B.1. R Studio	143
B.2. Visual Studio	143
C. Paquetes R destacados	145
C.1. <code>tidyverse</code>	145
C.2. <code>ggplot2</code>	145
C.3. <code>GGally</code>	145
C.4. <i>Pipelines</i>	145
C.5. <code>data.table</code>	145
C.6. Modelos estadísticos y aprendizaje automático	145
C.6.1. <code>tidymodels</code>	145
C.6.2. <code>mlr</code>	145
Referencias	147

Introducción

Bienvenidos al Taller de Introducción a R para estudiantes del Programa de Doctorado en Ciencias Agrarias y Ambientales de la UCLM.

En este taller se introducen los elementos básicos para programar con el lenguaje R. Este es un lenguaje de programación científica, creado originalmente para aplicaciones matemáticas y estadísticas y que está disponible como software libre, bajo licencia GNU GPL 3.

Por otra parte, este no es un taller de introducción a la programación. Si no estás familiarizado con conceptos básicos como variables, sentencias o comandos, control de flujo de ejecución, funciones, etc. podrás seguir los contenidos desde un punto de vista práctico, pero muy probablemente no comprendas del todo bien cómo funcionan los ejemplos que se muestran. En este caso, te recomendamos que te familiarices primero con los conceptos básicos de programación antes repasar estos contenidos. Puedes consultar el capítulo Capítulo 11 para encontrar referencias que te ayuden a aprender conceptos básicos de programación.

Este es un **taller práctico**, que presenta numerosos ejemplos de comandos y código ejecutable para aprender a programar con el lenguaje R. Nuestro objetivo no es realizar una presentación exhaustiva de todas las posibilidades que ofrece R, sino solo los elementos básicos para adquirir destreza a nivel elemental. A partir de aquí, deberías ser capaz de seguir aprendiendo sobre aspectos del lenguaje R y las opciones que ofrecen multitud de paquetes de expansión de funcionalidad publicados en el repositorio CRAN (consulta la sección Sección 1.5).

Introducción

Este manual que acompaña al Taller de Introducción a R ha sido elaborado con Quarto (<https://quarto.org>), una herramienta de creación de documentación científica compatible, entre otros lenguajes, con R. Para aprender más detalles sobre Quarto puedes visitar la completa guía disponible en <https://quarto.org/docs/guide/>. En concreto, la creación de libros como este utilizando Quarto está documentada en <https://quarto.org/docs/books>.

Licencia

A menos que explícitamente se indique lo contrario, los materiales proporcionados en este taller se publican bajo una licencia Creative Commons Atribución-Compartir-Igual (CC-BY-SA) versión 4.0. Se puede acceder a los términos legales y el texto completo de esta licencia mediante este enlace.

Parte I.

**Instalación y conceptos
básicos**

1. El lenguaje R

1.1. ¿Qué es R?

R es un lenguaje de programación especialmente pensado para el análisis y la visualización de datos. Fue creado por Ross Ihaka y Robert Gentleman como una versión libre del lenguaje estadístico S, creado en Bell Labs. Su uso está ampliamente extendido en investigación, docencia e industria. Empresas como Google, Facebook, Twitter, NYT, Pfizer, Santander, BBVA y Telefónica, entre otras, utilizan este lenguaje en sus desarrollos. Entre otras características, R cuenta con: - Capacidad para el almacenamiento y manipulación efectiva de datos. - Operadores para realizar cálculos sobre variables indexadas (Arrays), en particular matrices. - Una amplia colección de herramientas para el análisis de datos. - Multitud de opciones para el análisis gráfico de datos. - Como cualquier otro lenguaje de programación, también incluye sentencias condicionales, ciclos, funciones recursivas y posibilidad de entradas y salidas.

1.2. RStudio

RStudio es un entorno de programación con R creado por la compañía norteamericana Posit PBC. Este entorno de desarrollo cuenta con un intérprete de R que evalúa expresiones aritméticas, lógicas y llamadas a funciones o asignaciones. Algunas de estas expresiones (siempre y cuando sean correctas) pueden producir una salida visible que se puede almacenar

1. El lenguaje R

en alguna variable, en disco o en un fichero. También es posible leer y escribir ficheros de datos, así como generar y guardar gráficos, crear *scripts* (archivos con listados de comandos ejecutables escritos en R) o crear ficheros de informes.

1.3. ¿Cómo se mantiene R en la actualidad?

Por un lado está la **R Foundation**, una fundación sin ánimo de lucro que gestiona IPR. Por otro lado tenemos **RCore Team**, un grupo de personas con privilegios para cambiar el código base de R. Adicionalmente están los **R Contributors**, numerosos desarrolladores que han realizado diversas aportaciones al proyecto.

R Consortium: <https://www.r-consortium.org>.

Comunidad R Hispano (<http://r-es.org>): Organiza jornadas anuales (conferencias, talleres...). Grupos locales, meetups (<http://madrid.r-es.org>)

1.4. Ventajas e inconvenientes de R.

Dentro de las ventajas de R podemos destacar las siguientes: - Se trata de una herramienta de software libre, gratuito. - La comunidad que le da soporte es una comunidad sólida, con múltiples participantes del mundo académico y de la industria. - Es posible obtener soporte comercial mediante proveedores. - Ya ha conseguido suficiente masa crítica. - Podemos adaptarlo y personalizarlo mediante el desarrollo de funciones, paquetes o plantillas de documentos. - Casi cualquier innovación en estadística ya está implementada en R. - Facilita la creación y automatización de flujos de trabajo y análisis reproducibles, utilizando herramientas adicionales como Quarto.

A pesar de todas sus bondades, podemos encontrar algunos inconvenientes: - Hay que aprender a programar (aunque solo sea un poco). - Para aquellos que saben programar en otros lenguajes, sobre todo siguiendo el paradigma de orientación a objetos, el lenguaje R no resulta intuitivo, es decir, presenta una cierta curva de aprendizaje. - Existen interfaces gráficas, pero es importante acostumbrarse a revisar el código fuente y la salida. Solo así comprendemos qué hace realmente el método o algoritmo que estamos aplicando.

1.5. Paquetes R

Una de las características más potentes de R es la disponibilidad de una enorme cantidad de **paquetes** que amplían su funcionalidad. Podemos ver los paquetes R como módulos que podemos conectar dinámicamente a nuestros programas para disponer de inmediato de un nuevo catálogo de funciones y elementos para realizar diversas tareas: trabajar con nuevos tipos de modelos estadísticos o de aprendizaje automático, lectura y escritura de datos, preparación y limpieza de datos, etc.

En la página web <https://cran.r-project.org/>, si pulsamos en el enlace *Packages* en el menú lateral izquierdo se mostrará una página que resume información sobre los paquetes R disponibles para ampliar la funcionalidad de nuestros programas. En este momento, existen más de 21.400 paquetes publicados, lo que da una idea del vasto catálogo de herramientas que la comunidad R pone a nuestra disposición.

Una ventaja que conviene resaltar es que muchos de estos paquetes han sido desarrollados por destacados/as científicos/as y profesionales de diferentes ámbitos (Matemáticas, Estadística, Ciencias de la Computación), así como de numerosas áreas de aplicación, entre ellas las Ciencias Agrarias y Medioambientales. Para cada paquete, existe una página de descripción en la que podemos consultar los datos sobre las personas que lo han creado y las encargadas de mantenerlo.

1. El lenguaje R

Para poder utilizar un paquete primero debemos instalarlo (sólo **una vez**), mediante el comando `install.packages("nombre-del-paquete")`. Por ejemplo, para instalar el paquete `dplyr` ejecutamos:

```
install.packages("dplyr")
```

Una vez que el paquete se instala sin errores, a partir de ese momento podemos ya cargarlo directamente en cualquier *script* o programa de R mediante el comando `library`:

```
library(dplyr)
```

Después de ejecutar el comando, todas las funciones y resto de elementos (como conjuntos de datos) contenidos en el paquete pasan a estar disponibles para ser utilizados en nuestros programas.

Orden de carga de los paquetes

Debemos ser cuidadosos al decidir en qué orden vamos a cargar paquetes adicionales en nuestro *script* o programa de R. Alguno paquetes pueden tener elementos (funciones, objetos, *datasets*, etc.) cuyo nombre coincida con el de otro elemento previamente cargado en un paquete anterior.

En estos casos de coincidencia de nombres, ocurre que el último elemento cargado sobrescribe a otro del mismo nombre cargado desde un paquete anterior. En estos casos, podemos fijarnos en que el intérprete de R nos avisará mediante mensajes de qué elementos han sustituido a otros con el mismo identificador cargados con anterioridad.

Una solución para indicar exactamente qué elemento queremos invocar y a qué paquete pertenece es utilizar la sintaxis `paquete::funcion()`.

1.6. Fuentes de información sobre R

- Cuenta de Posit PBC en Instagram.
- Canal Posit PBC en Youtube.
- Grupos locales de usuarios R.
- Meetups sobre R

2. Instalación de R y R Studio

R y RStudio pueden ser instalados sobre plataformas Windows, MAC OSX y Linux:

- R se puede descargar e instalar desde la “Comprehensive R Archive Network” (CRAN). Enlace: <https://cran.rediris.es/>.
- Una vez instalado R, puedes descargar e instalar RStudio desde: <http://www.rstudio/products/RStudio/>.
 1. Selecciona la plataforma de ejecución deseada
 2. Descargamos el archivo instalador para nuestra plataforma.
 3. Accedemos a la carpeta donde se haya descargado el archivo y procedemos a su instalación.

Ahora ya podremos ejecutar RStudio y comenzar a trabajar en nuestros proyectos de programación con R.

2.1. Trabajando con RStudio

La pantalla principal de RStudio se divide en cuatro áreas o **paneles**, cuya posición y tamaño podemos modificar:

2. Instalación de R y R Studio

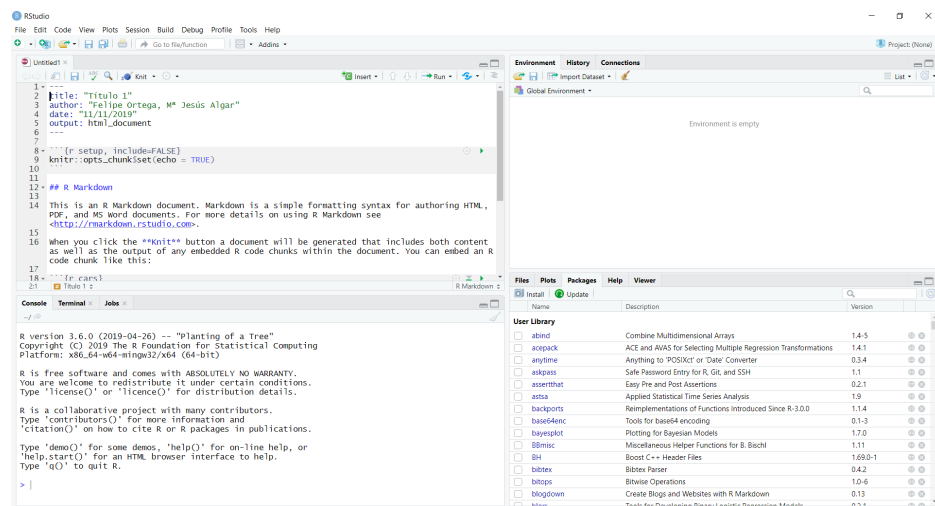


Figura 2.1.: Fig. 1. Interfaz gráfica de RStudio.

El panel superior izquierdo es el *editor de código*, donde se escriben los programas. Nuestro código podemos escribirlo en scripts, con el fin de poder utilizarlos en otros proyectos. En estos scripts se realiza un diagnóstico de sintaxis con el siguiente código de colores: una cruz roja indica que hay un error de sintaxis, una exclamación amarilla indica que la variable o el objeto no se utiliza o que no se ha declarado, y un círculo azul nos ofrece la opción de solicitar información de ayuda sobre lo que estamos haciendo.

Justo debajo del editor de código se encuentra el panel de la *consola*, que es la parte principal de RStudio. Se trata de un intérprete interactivo de comandos en el que podemos escribir líneas de código. Las líneas de código pueden ser:

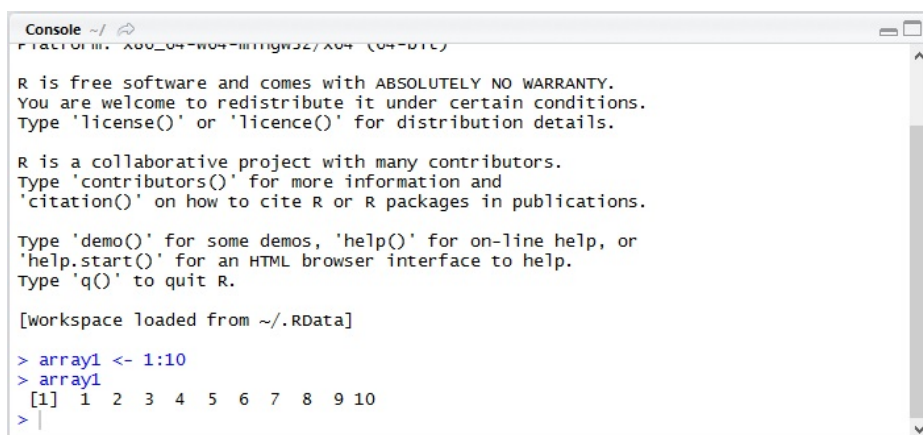
- Órdenes elementales (instrucciones) como una expresión o una asignación. Una expresión se evalúa, se imprime su valor y se pierde, es decir, no se puede utilizar posteriormente. Sin embargo, en una

2.1. Trabajando con RStudio

asignación, se evalúa la expresión, no se imprime el valor y éste se guarda en una variable, para poder ser utilizado posteriormente.

- Solicitudes de ayuda, por ejemplo: `help(solve)` o, de forma alternativa, `?solve`.
- Llamadas a funciones.
- Instrucciones que generen gráficas.

Si la sentencia produce una salida, al darle a intro esta salida se visualizará justo debajo de la línea de código y se habilitará la siguiente línea para introducir la siguiente instrucción, tal y como se observa en la siguiente figura.



```
Console ~/...
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[workspace loaded from ~/.RData]

> array1 <- 1:10
> array1
[1] 1 2 3 4 5 6 7 8 9 10
> |
```

Figura 2.2.: Fig. 2. Línea de código en la consola.

Como usuarios principiantes de R, podemos ver la consola como el espacio en el que vamos a teclear todas nuestras líneas de código. Sin embargo, conforme vayamos aconstumbrándonos al lenguaje y al entorno, nuestras líneas de código pasarán a estar en scripts que podremos abrir, ejecutar y modificar, tantas veces como queramos.

En la parte superior derecha está el panel de *programación* donde apare-

2. Instalación de R y R Studio

cen utilidades como el listado de las variables de entorno, el historial de comandos previamente ejecutados, acceso a herramientas para control de versiones de código o conexiones a fuentes externas de datos (bases de datos, etc.). En la pestaña de *Environment* por cada objeto que creamos en nuestro código, su contenido lo podemos ver aquí de forma automática. Se pueden guardar los objetos del espacio de trabajo en un fichero con extensión `.RData`. También es posible recuperar los objetos creados en alguna sesión anterior sin más que cargar el correspondiente fichero con extensión `.Rdata`. Además, es posible limpiar los objetos que hay en el espacio de trabajo. Al conjunto de objetos que tengamos almacenados en cada momento se denomina *espacio de trabajo (workspace)*.

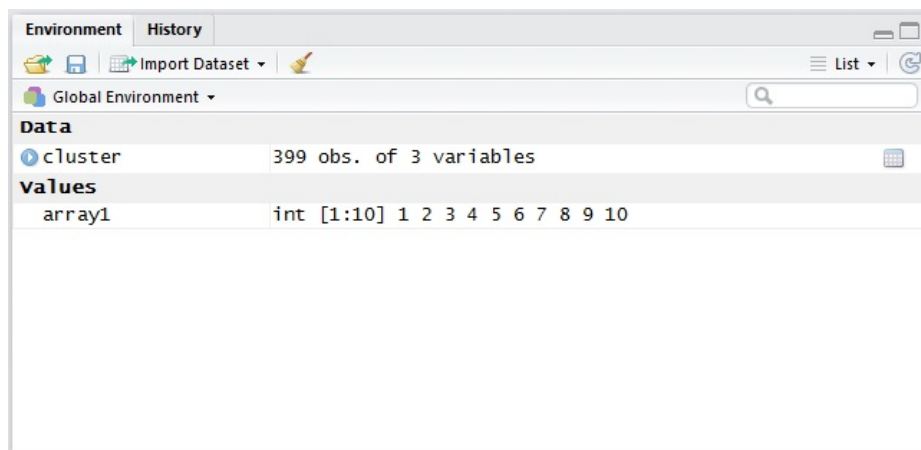


Figura 2.3.: Fig. 3. Environment.

Una opción muy importante es la poder importar un Dataset. Después de elegir el formato en el que se encuentra el Dataset y su localización (ruta), en la ventana de *código* se abrirá una nueva pestaña en la que podremos ver el contenido del Dataset.

La pestaña *History* muestra el historial de todos los comandos introducidos. Toda instrucción que escribamos en la línea de comandos se carga

2.1. Trabajando con RStudio

aquí automáticamente. Se puede guardar el contenido de esta ventana en un fichero, con extensión `.Rhistory`. Por lo que, también podremos cargar (abrir) un fichero de históricos. Nos ofrece la posibilidad de poder recuperar las instrucciones y pasarlas a la consola o al script que tengamos abierto. Mediante el icono de la escoba, podremos borrar todo el histórico de instrucciones.

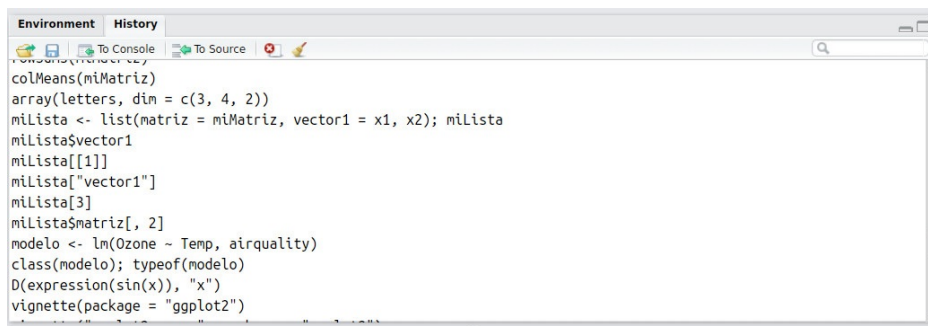


Figura 2.4.: Fig. 4. History.

Por último, en la parte inferior derecha se encuentra el panel *multifunción*, donde está el navegador de archivos, el carrusel de gráficas, el gestor de paquetes, la ayuda en línea y el visualizador. En este panel podremos realizar diversas actividades: - Mediante la pestaña *Files* podemos explorar archivos en nuestro ordenador. - En la pestaña *Plots* irán apareciendo los diversos gráficos que vayamos construyendo. Podremos navegar a través de todas las representaciones que hayamos hechos, podremos aplicar zoom en ellas, exportarlas en diferentes formatos. También podremos eliminar individualmente las representaciones que tengamos o, incluso, eliminarlas todas. - En *Packages* podremos ver todos los paquetes que tenemos instalados en RStudio. Mediante el icono *install* podremos instalar nuevos paquetes, sin más que seleccionar el repositorio, tecleando el nombre del paquete e indicando la ruta donde se va a instalar. También se pueden actualizar los paquetes instalados y eliminar los que ya no nos interesen.

2. Instalación de R y R Studio

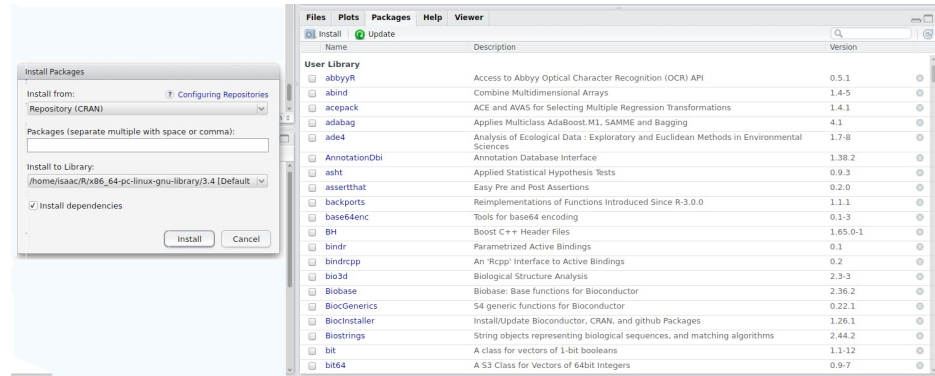


Figura 2.5.: Fig. 5. Packages e instalación de paquetes.

- Cuando se ejecute una instrucción que solicite ayuda sobre un determinado comando, esta ayuda se visualizará en la pestaña *Help*. También nos permite navegar a través de todos los topics de ayuda que hayamos realizado, así como poder imprimirlos. Incluso, es posible realizar una búsqueda de ayuda general. Para cualquier documento que estemos consultando, tiene un buscador para buscar dentro de él.

2.1. Trabajando con RStudio

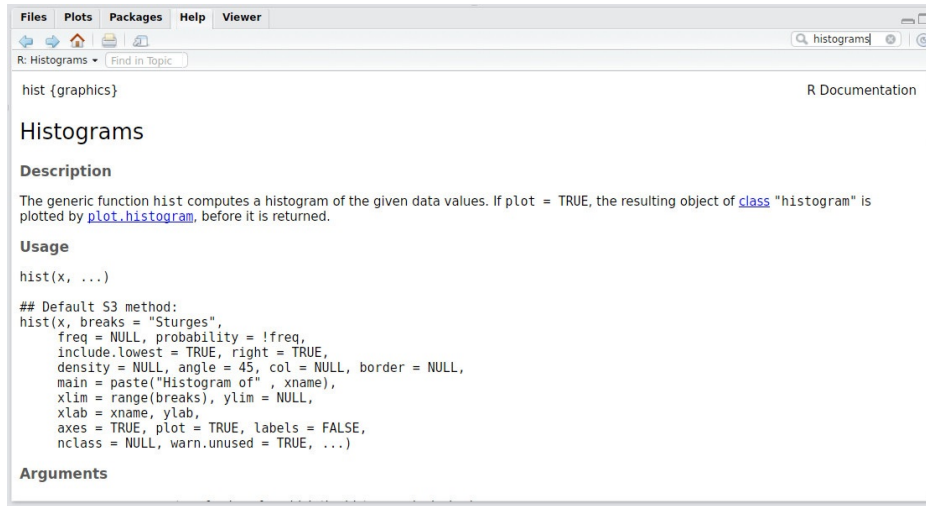


Figura 2.6.: Fig. 6. Help.

A continuación ofrecemos un resumen de comandos rápidos para trabajar en RStudio.

Descripción	Comando
Navegar por el histórico de comandos para seleccionar el que queremos	Ctrl +
Autocompletado (dentro de los paréntesis de una función nos mostrará los parámetros de la función)	Tabulador
Simular código seleccionado	Ctrl + Intro
Mover el cursor a la Console	Ctrl + 2
Mover el curso al Source	Ctrl + 1
Incluir comentario	Ctrl + shift + C
Undo	Ctrl + Z
Redo	Ctrl + shift + Z

Figura 2.7.: Fig. 7. Comandos rápidos.

2.2. Proyectos de RStudio y directorio de trabajo

Los diferentes proyectos que vayamos haciendo los iremos guardando en diferentes directorios, esto es, en diferentes carpetas. En cada uno de estos directorios estarán localizados el conjunto de scripts y de paquetes que se usan en ese proyecto, los datos de entrada y los resultados analíticos y representaciones que hayamos hecho. A esto se le llama directorio de trabajo. En el menú *Session* tenemos la opción de establecer un determinado directorio como directorio de trabajo:

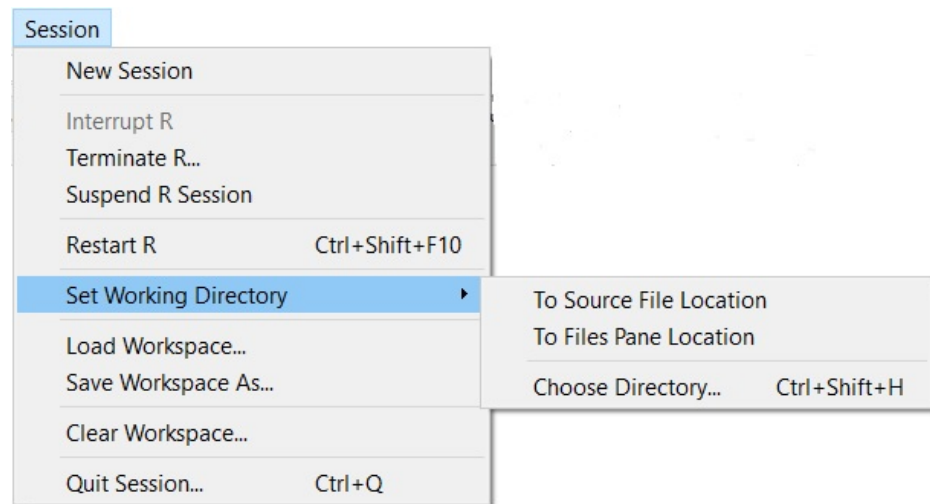


Figura 2.8.: Fig. 8. Selección del directorio de trabajo.

Veamos, mediante un sencillo ejemplo, como se trabaja con proyectos en RStudio. Lo primero es crear el proyecto, en el menú *File* tenemos la opción *New Project*:

2.2. Proyectos de RStudio y directorio de trabajo

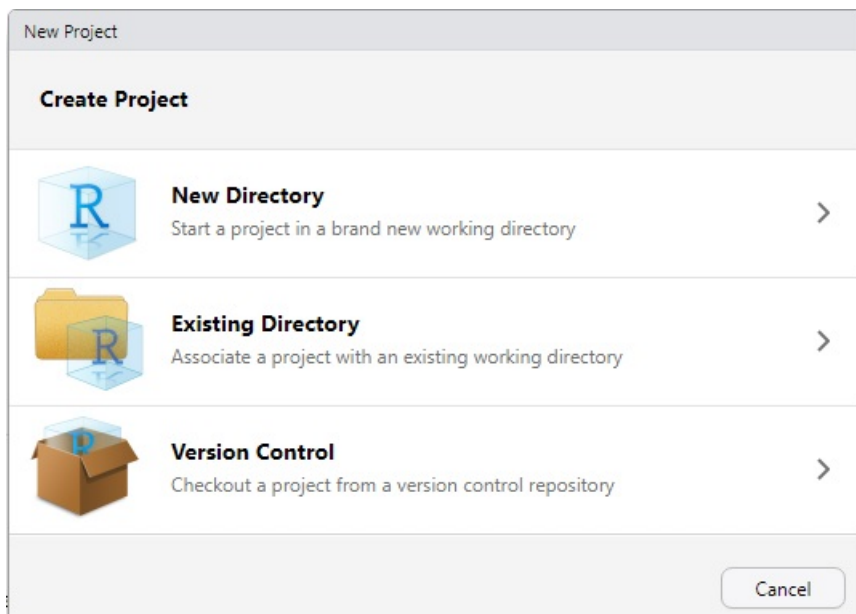


Figura 2.9.: Fig. 9. Paso 1: ventana New Project.

Haciendo click en *New Directory*, llegaremos a la siguiente ventana donde debemos seleccionar la opción *New Project*:

2. Instalación de R y R Studio

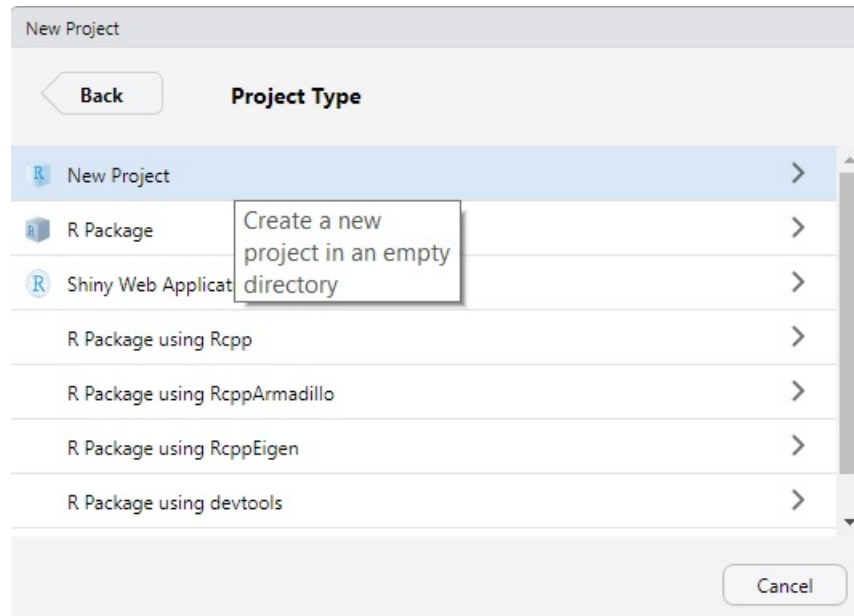


Figura 2.10.: Fig. 10. Paso 2: ventana New Directory.

Finalmente indicamos el nombre que queremos darle a nuestro proyecto, por ejemplo: *Poryecto1*. Haciendo click en *Create Project* ya tendremos nuestro primer proyecto creado.

2.2. Proyectos de RStudio y directorio de trabajo

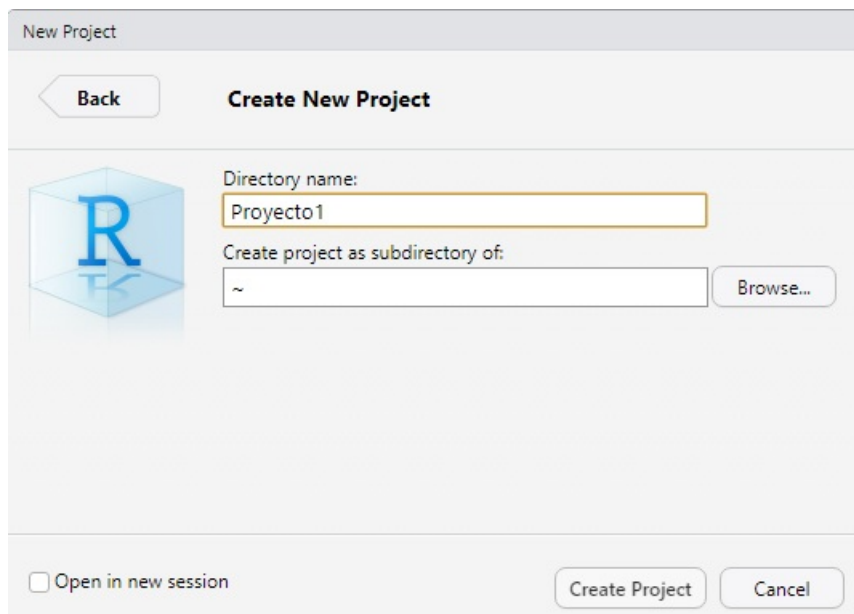


Figura 2.11.: Fig. 11. Paso 3: nombre del proyecto.

Ahora estamos en disposición de poder introducir nuestras líneas de código. Para ello, copiaremos las siguientes instrucciones en la consola. Conforme vayamos introduciendo las líneas de código, en la pestaña Environment irán apareciendo cada uno de los objetos que vamos creando junto con sus valores. Cuando introduzcamos las instrucciones que generan una representación gráfica, ésta aparecerá en la pestaña Plots.

2. Instalación de R y R Studio

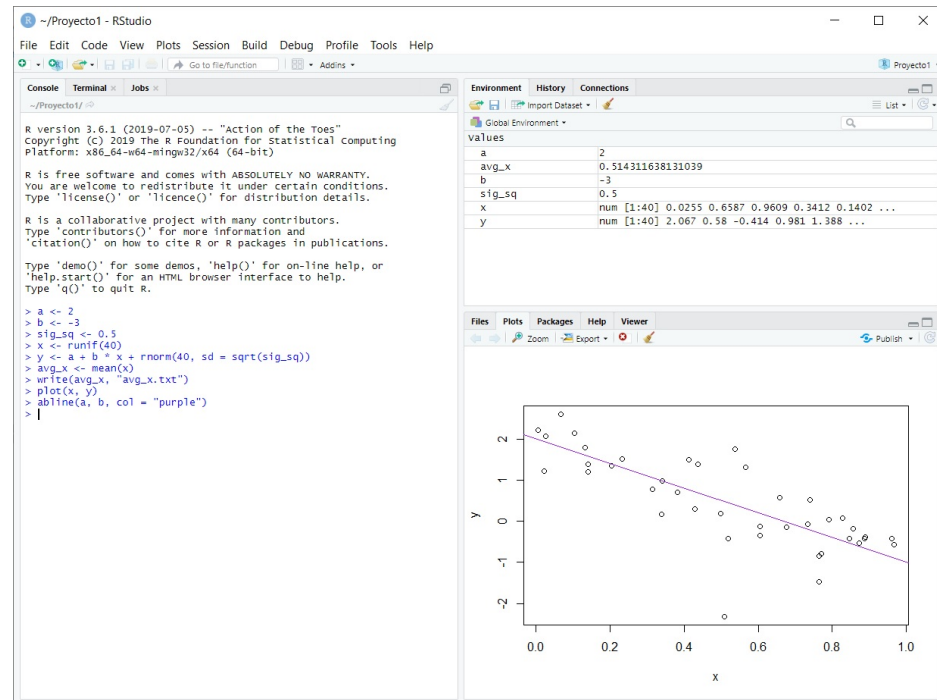


Figura 2.12.: Fig. 12. Paso 4: código del proyecto.

Si en la pestaña *History* pulsamos el icono *Guardar* y salvamos el código en un fichero con extensión *.R* (es decir, en un *script*), cuando volvamos abrir Rstudio podremos abrir el código de nuestro proyecto y modificarlo tantas veces como queramos. Más adelante (en otro taller) abordaremos alternativas más avanzadas para generar documentos que combinen contenido textual formateado con bloques de código ejecutable R, que pueden generar también resultados (numéricos, gráficos, tablas) que se integran directamente en el documento.

2.2. Proyectos de RStudio y directorio de trabajo

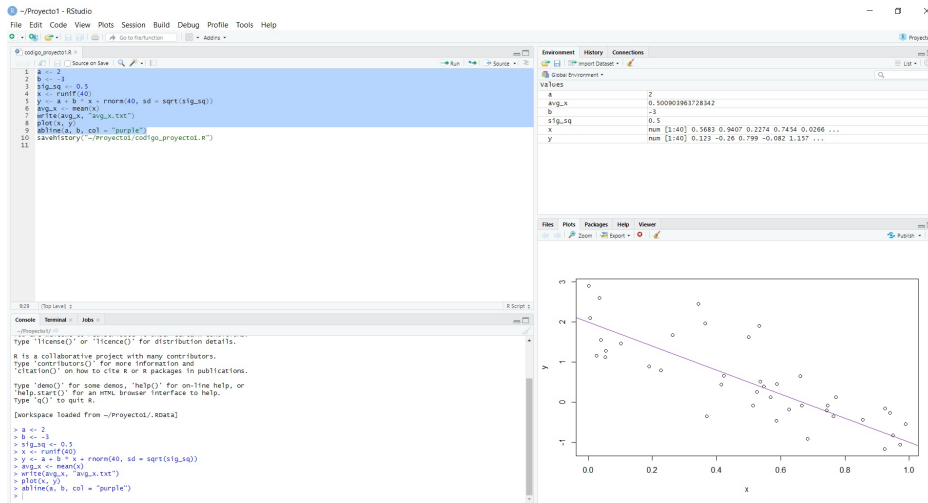


Figura 2.13.: Fig. 13. Abrir el proyecto.

La creación de proyectos nos facilita el trabajo en RStudio, ya que nos permite tener juntos todos los ficheros de código, almacenar el histórico de comandos y guardar las variables de entorno, tal y como las dejamos al cerrar la sesión.

3. Organización de proyectos

Los **proyectos** nos permiten trabajar de manera más efectiva en RStudio. Un proyecto mantiene información sobre todos los elementos de programación que necesitamos (el espacio de trabajo, el directorio de trabajo, las variables de entorno, etc.). De este modo, podemos cerrar el proyecto y al volver a abrirlo todo estará preparado para continuar desde el momento en que lo dejamos.

En la sección Sección 2.2 ya se ha explicado paso a paso cómo podemos crear un nuevo proyecto con RStudio.

3.1. Estructura de directorios

Además de crear un proyecto, resulta conveniente seguir un esquema consistente para la creación de carpetas dentro del directorio principal de nuestro proyecto. De este modo, todo estará siempre bien organizado y cuando acudamos a un proyecto que se creó hace tiempo nos resultará más sencillo encontrar rápidamente los recursos que buscamos.

No se puede dar una fórmula general para un listado de carpetas válido para cualquier proyecto, pero si se pueden sugerir carpetas que suelen ser útiles. Cada programador o equipo debe decidir, en función del tipo de proyectos, los recursos a gestionar y sus objetivos, qué combinación de carpetas es más adecuada en cada caso.

3. Organización de proyectos

- **data**: directorio para almacenar los archivos de datos disponibles. Por ejemplo, datos que hemos descargado previamente o que se descargarán periódicamente de forma automatizada mediante herramientas o *scripts*, datos generados por nuestros programas, etc.
- **R** o **r**: carpeta para guardar nuestros *scripts*. Incluso en el caso en que sólo tengamos un *script* en nuestro proyecto, es recomendable crearla porque es muy probable que el proyecto crezca y terminemos creando más (se debe evitar a toda costa crear *scripts* “kilométricos”).
- **doc**: para documentación de nuestro proyecto, ejemplos de uso, etc. En el caso de que usemos una herramienta de documentación específica habrá que leer su manual de uso para seguir exactamente la estructura que necesite.
- **deliverables** o **inform**: para contener informes o documentos de resumen creados con Quarto o RMarkdown con extractos y resúmenes de resultados.
- **slides** o **diapos**: si tenemos que preparar presentaciones con diapositivas puede ser una buena idea organizarlas en su propio directorio independiente.
- **img**: si nuestro proyecto emplea imágenes (mapas, tomas satelitales, fotografías, etc.) es una buena idea organizarlas todas dentro de una misma carpeta, aparte del resto de datos.
- **test**: puede ser útil para almacenar *scripts* de prueba que todavía no son del todo correctos o no están probados a fondo. También para incluir pruebas formales de corrección de nuestros *scripts* (comprobación de robustez).
- **tmp**: para almacenar archivos de datos intermedios o ficheros efímeros, que no pretendemos conservar más que para unas pocas ejecuciones (pruebas, etc.).

3.2. Tipos de proyectos

- **log**: si nuestro proyecto involucra una gran cantidad de operaciones que pueden consumir mucho tiempo de ejecución, es conveniente que se generen ficheros de registro o *logs* del progreso de dichas operaciones.
- **util o utils**: conserva *scripts* que suelen contener funciones y otras herramientas de utilidad que se emplean para realizar operaciones básicas en los *scripts* principales del directorio R. De esta forma, se *modulariza* el código, separándolo en ficheros diferentes para gestionarlo y mantenerlo mejor (recordemos que hay que evitar meter todo en un solo archivo enorme).
- **aux, notas, utils, labs, practicas, ejercicios, etc.**: posibles directorios que pueden surgir en función del tipo de proyecto que abordemos (prácticas de un curso, experimentos, laboratorios y otros).

3.2. Tipos de proyectos

- Proyecto estándar de análisis de datos.
- Paquete R.
- Sitio web.
- Libros/documentos científicos.
- Presentación con diapositivas.

Parte II.

El lenguaje R

4. Programación básica en R

4.1. Objetos y variables.

En este apartado se ofrece un breve repaso a la sintaxis básica de R.

Los objetos o variables de R pueden almacenar información de diferentes tipos:

- *valores numéricos* en variables de tipo numérico.
- *valores alfabéticos* en variables de tipo carácter, que en realidad son cadenas de caracteres que van entre comillas dobles o comillas simples.
- *valores lógicos* en variables de tipo booleano, que solo pueden tomar los valores TRUE o FALSE.

Cuando se crea una variable, la asignación de un valor a esta variable puede hacerse de la siguiente manera:

```
vat1 <- 1.2  
vat1
```

```
[1] 1.2
```

```
vat2 = 0.25  
vat2
```

4. Programación básica en R

```
[1] 0.25
```

En este ejemplo puede verse como se han creado dos variables de tipo numérico utilizando dos símbolos diferentes para realizar la asignación.

Cualquier variable solo puede tener un único valor en cada instante de tiempo. De modo que si se asigna un nuevo valor a una variable, estamos eliminando el valor que habíamos almacenado antes.

```
y = 93  
y
```

```
[1] 93
```

```
y = 52  
y
```

```
[1] 52
```

La asignación también se puede hacer mediante una expresión. Esta expresión se evaluará y el valor obtenido se asigna al objeto:

```
z = 3  
w = z^2  
w
```

```
[1] 9
```

```
i = (z*2+75)/2  
i
```

```
[1] 40.5
```

Por lo tanto, la asignación es una operación en la que se evalúa la expresión que hay a la derecha y se le asigna ese valor a la variable de la izquierda.

4.2. Funciones.

4.1.1. Función `ls()`.

Muestra una lista de todos los objetos o variables que se han creado.

4.1.2. Función `rm()`.

Libera el espacio de memoria que ocupa el objeto o variable que hayamos indicado entre paréntesis.

4.1.3. Operadores aritméticos.

Los operadores aritméticos elementales son los habituales `+`, `-`, `*`, `/` y `^` para elevar a una potencia. Además están disponibles las funciones `log`, `exp`, `sin`, `cos`, `tan` y `sqrt`.

4.1.4. Operadores lógicos.

Los operadores lógicos son: `<` (menor), `<=` (menor o igual), `>` (mayor), `>=` (mayor o igual), `==` (igual), y `!=` (distinto). Si `v1` y `v2` son dos expresiones lógicas, entonces `v1&v2` es su intersección (“conjunción”), `v1|v2` es su unión (“disyunción”) y `!v1` es la negación de `V1`.

4.2. Funciones.

Las funciones en R son un tipo especial de objeto diseñado para llevar a cabo alguna operación. Se aplican a un conjunto de argumentos y producen un resultado.

R tiene una amplia variedad de funciones que el usuario puede utilizar. Pero también puede crearse las suyas propias. En el caso de querer utilizar una función R, no hay más que llamarla escribiendo su nombre y su lista

4. Programación básica en R

de argumentos (en caso de tenerlos). La lista de argumentos siempre va entre paréntesis y separados por comas.

```
# Esta instrucción devuelve el máximo de todos los números que se le han pasado  
max(4,5,6,12,-4)
```

```
[1] 12
```

```
# Máximo de una secuencia de 30 elementos elegidos al azar de entre el 1 y el 100  
max(sample(1:100, 30))
```

```
[1] 98
```

Al igual que en otros lenguajes de programación, resulta interesante implementar una función cuando necesitas que varias líneas de código se ejecuten varias veces en diferentes puntos de nuestro código. En este caso, ese bloque de código lo encapsulamos dentro de una función, la cual llamaremos cada vez que la necesitemos. Antes de darle un nombre a una función que implementemos nosotros, podemos comprobar si ese nombre no se ha utilizado ya, mediante la función `exists()` (devuelve un TRUE o un FALSE en función de si el nombre de la función ya existe o no, respectivamente).

La forma de implementar una función es la siguiente:

```
nombreFuncion = fuction(argumento1, argumento2,...){ instrucción1 instrucción2 . . . return resultado }
```

El cuerpo de la función está delimitado por `{}` y cada instrucción debe ocupar una línea

4.2. Funciones.

```
# Función que calcula el error estandar (raiz cuadrada de la varianza entre el tamaño de la
se = function(x){
  v = var(x)
  # var(): función de R que devuelve la varianza de un conjunto de valores, en este caso, de
  n = length(x)
  # length(): función de R que devuelve el número de valores en x
  return(sqrt(v/n))
  # sqrt(): función de R que devuelve la raiz cuadrada
}
```

Una vez que hemos creado la función, podemos usarla:

```
# rnorm(): función de R que devuelve una muestra de 100 números aleatorios tomados una distr
mySample = rnorm(100, mean=20, sd=4)
se(mySample)
```

```
[1] 0.3732302
```

4.2.1. Parámetros con valores por defecto.

Si alguno de los parámetros de nuestra función tiene que tomar algún valor por defecto debemos indicarlo en la cabecera de la función.

```
#Función que convierte un valor en km, decímetros, centímetros o milímetros a metros.
convUnitsToMeters = function(val, from="km"){
  #La función switch de R compara el contenido de una variable (en este caso "to") con
  #un conjunto de opciones. En el caso de que la variable indicada no se corresponda con
  #ninguna de las opciones posibles, se retornará el valor especial NA
  mult = switch(from, km=1000, dm=0.1, cm=0.01, mm=0.001, NA)
  if (is.na(mult)) stop("Unknown target unit of length")
  else return (val*mult)
```

4. Programación básica en R

```
}  
#Si se llama a la función con dos parámetros, el valor por defecto se sustituye  
#por el valor indicado por el usuario  
convUnitsToMeters(23,"km")
```

```
[1] 23000
```

```
convUnitsToMeters(40,"cm")
```

```
[1] 0.4
```

```
#Si solo se pasa un parámetro, el segundo toma el valor por defecto  
convUnitsToMeters(40)
```

```
[1] 40000
```

4.2.2. Llamada a una función.

La llamada a una función se puede realizar de dos formas: - Indicando el nombre de los parámetros. En este caso el valor que toma cada parámetro es el indicado. La ventana que ofrece el llamar a una función indicando el nombre de los parámetros, es que podemos omitir el nombre de parámetros que tengan valor por defecto y no queramos modificar ese valor e indicar solo aquel parámetro o parámetros que si queramos modificar.

```
convUnitsToMeters(from="km",val=56.2)
```

```
[1] 56200
```

4.2. Funciones.

- Sin indicar el nombre de los parámetros. En este caso el valor de los parámetros se susittye por posición. En el ejemplo que se muestra a continuación, el 1º argumento toma el valor 56.2 y el segundo toma el valor “yard”.

```
convUnitsToMeters(56.2, "km")
```

```
[1] 56200
```

También es posible mezclar ambas formas de llamar a una función:

```
convUnitsToMeters(56.2, from="km")
```

```
[1] 56200
```


5. Tipos de datos

5.1. Vectores.

Es el objeto básico de R. Incluso cuando estamos asignando un valor a una variable, en realidad lo que estamos haciendo es un vector de un solo elemento. Como en cualquier otro lenguaje de programación, un vector es una estructura en la que podemos almacenar valores del mismo tipo (numéricos, valores lógicos o cadenas de caracteres), es decir, es una estructura de datos **homogénea** en la que los valores almacenados ocupan una determinada posición.

5.1.1. Creación de vectores.

Existen varias formas de definir un vector:

- Mediante el operador “:”. Es una forma sencilla de generar secuencias de valores. Este operador tiene la máxima prioridad en una expresión.

```
v = 1:10  
v
```

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

5. Tipos de datos

```
# Hay que tener cuidado con la precedencia del operador :  
10:15 - 1
```

```
[1]  9 10 11 12 13 14
```

```
10:(15 - 1)
```

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

```
# Secuencia decreciente  
5:0
```

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

- Una forma fácil de crear un vector en R es mediante la función **c()**, a la que se le indica entre paréntesis los elementos que va contener el vector. El número de elementos de un vector es su tamaño y podemos obtenerlo mediante la función **length()**.

```
v = c(5, 2, 53.85, 43.2, 30)  
v
```

```
[1] 5.00 2.00 53.85 43.20 30.00
```

```
length(v)
```

```
[1] 5
```

5.1. Vectores.

```
mode(v) #Indica el tipo de datos que almacena el vector.
```

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

```
v = c(5, 2, 53.85, 43.2, 30, "abc")
#Todos los elementos del vector se han convertido a tipo string (lo comprobamos).
v
```

```
[1] "5"      "2"      "53.85"  "43.2"   "30"     "abc"
```

```
mode(v)
```

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

- Y mediante la función **seq()**. Esta función tiene cinco argumentos, aunque no se utilizan todos simultáneamente. Si se dan los dos primeros indican el comienzo y el final del vector. Si solo se facilitan estos dos argumentos, el resultado coincide con el operador ‘dos puntos’.

```
x = seq(-3, 3, by=.5) #inicio, fin, salto
x
```

```
[1] -3.0 -2.5 -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3.0
```

```
# Al especificar que solo queremos 4 elementos, calcula él el salto
seq(from = 1, to = 5, length = 4)
```

```
[1] 1.000000 2.333333 3.666667 5.000000
```

5. Tipos de datos

```
# Ahora le especificamos el tamaño, el inicio y el salto, por lo que calcula  
# cual será el fin  
seq(length = 10, from = -2, by = 0.2)
```

```
[1] -2.0 -1.8 -1.6 -1.4 -1.2 -1.0 -0.8 -0.6 -0.4 -0.2
```

- Otra forma frecuente de generar secuencias es mediante la función `rep()`:

```
rep(5, 4)
```

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

```
rep(1:2, 3)
```

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

```
rep(1:2, each = 3)
```

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

- R también tiene diversas funciones para generar secuencias aleatorias. Todas ellas siguen esta estructura: `rfunc(n, par1, par2, ...)`, donde *func* es el nombre de la distribución de probabilidad, *n* es el número de elementos que se van a generar, y *par1*, *par2*, ... son el resto de parámetros propios de cada tipo de distribución de probabilidad.

```
# Distribución normal  
rnorm(10)
```


5.1. Vectores.

```
[1] -0.46621926  0.01455306  0.42916396  0.43606135 -0.46743003  0.53042649  
[7]  0.06015889 -0.79257149 -0.55419068  0.72357552
```

```
rmnorm(4, mean=10, sd=3)
```

```
[1]  9.578814 10.894008 10.964702  8.503657
```

```
# Distribución t Student  
rt(5, df=10)
```

```
[1]  1.2092747 -0.3536291  0.1404271 -0.8091649  0.2006938
```

- La función **vector()** nos permite crear vectores vacíos.

```
x = vector()  
x
```

```
logical(0)
```

5.1.2. Valores faltantes.

En determinadas ocasiones puede que no todas las componentes de un vector sean conocidas. Cuando falta un elemento, es decir, tenemos un ‘valor faltante’, se le asigna un valor especial: **NA**. En general, cualquier operación donde intervenga un valor **NA** da por resultado **NA**. Este resultado es lógico, ya que la operación no se puede realizar porque los datos son incompletos. Existe la función **is.na(x)** que crea un vector lógico del tamaño del vector **x** que se le pasa por parámetro cuyos elementos valdrán **TRUE**, si el elemento correspondiente de **x** es **NA**, y **FALSE**, en caso contrario.

5. Tipos de datos

```
v1 = c(FALSE, TRUE, NA, TRUE, TRUE)
v1
```

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

Algunas funciones de gran utilidad al a hora de trabajar con vectores son:

- **min()** y **max()** que seleccionan respectivamente el menor y el mayor valor de sus argumentos, incluso cuando el argumentos son varios vectores,
- **range()** cuyo valor es el vector de longitud dos: `c(min(x), max(x))`,
- **sum(x)** que es la suma de todos los elementos de `x`,
- **prod(x)** que es el producto de todos ellos, y
- **sort(x)** que devuelve un vector del mismo tamaño que `x` con los elementos ordenados en orden creciente.

5.1.3. Indexación de elementos

Se puede acceder a un elemento en particular mediante el índice de la posición que ocupa.

```
# Elemento que se encuentra en la 1ª posición
v1[1]
```

```
[1] FALSE
```

```
# Elemento que se encuentra en la 5ª posición
v1[5]
```

```
[1] TRUE
```

5.1. Vectores.

```
# OJO: en R, el acceder a una posición que NO existe no genera ningún error  
v1[7]
```

```
[1] NA
```

El tamaño de un vector se puede modificar añadiendo más elementos.

```
# x era un vector con elementos de -3.0 a 3.0 con salto de 0.5  
# Añadimos el elemento 28 en la posición 15:  
x[15] = 28  
# Como en la posición 14 no tenía nada, se ha añadido un NA, hasta llegar a la nueva posición  
x
```

```
[1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA 28
```

Podemos seleccionar determinados valores de un vector atendiendo a una condición lógica. A esta condición o expresión lógica se la denomina **máscara**. También podemos ver esta opción como otra forma de construir un nuevo vector a partir de uno existente.

```
x = c(0, -3, 4, -1, 45, 90, -5)  
x[x>0]
```

```
[1] 4 45 90
```

```
x[x<=-2 | x>5]
```

```
[1] -3 45 90 -5
```

5. Tipos de datos

```
x[x>40 & x<100]
```

```
[1] 45 90
```

La máscara también puede ser un vector de enteros, de manera que se seleccionan aquellos elementos que están en esas posiciones:

```
x[c(4,6)]
```

```
[1] -1 90
```

```
x[1:3]
```

```
[1] 0 -3 4
```

Si los índices de las máscaras son negativos, cambia el significado de la máscara en el sentido en que lo que se está indicando son los índices que no se quieren coger.

```
x[-1]
```

```
[1] -3 4 -1 45 90 -5
```

```
x[-c(4, 6)]
```

```
[1] 0 -3 4 45 -5
```

5.1. Vectores.

```
x[-(1:3)]
```

```
[1] -1 45 90 -5
```

Como R te permite darle nombre a los elementos de un vector, la máscara también puede estar formada por un vector de string, de manera que en ella se nombre solo a los elementos que se quieren.

```
pH = c(4, 5, 7, 7.3, 8.2, 6.3)
names(pH) = c("area1", "area2", "mud", "dam", "middle")
pH
```

```
area1 area2 mud dam middle <NA>
  4.0   5.0  7.0  7.3   8.2   6.3
```

```
pH2 = c(area1=4.5, area2=7, mud=7.3, dam=8.2, middle=6.3)
pH2["mud"]
```

```
mud
7.3
```

```
pH2[c("area1", "dam")]
```

```
area1 dam
  4.5   8.2
```

5. Tipos de datos

5.1.4. Vectorización.

Uno de los aspectos más importantes de R es la vectorización de varias de sus funciones, es decir, la operación se realiza elemento a elemento. Estas funciones pueden ser aplicadas directamente a un vector generando otro vector resultado del mismo tamaño.

```
v = c(4, 7.5, 15, 76.2, 78)
# La función sqrt() calcula la raíz cuadrada de su argumento. Al pasarle un v
# genera otro vector del mismo tamaño que resulta de hacer la raíz cuadrada d
# elementos.
sqrt(v)
```

```
[1] 2.000000 2.738613 3.872983 8.729261 8.831761
```

```
v1 = c(1, 3, 5)
v2 = c(18, 9, 28)
v1+v2
```

```
[1] 19 12 33
```

```
2*v1
```

```
[1] 2 6 10
```

Si los vectores no son del mismo tamaño, R repite el vector pequeño tantas veces como sea necesario hasta rellenar todas las posiciones del grande:

```
v1 = c(4, 6, 8, 24)
v2 = c(10, 2)
v1 + v2
```

5.2. Factores.

```
[1] 14  8 18 26
```

```
# Es como si v2 fuera (10, 2, 10, 2)
```

Si el tamaño del vector pequeño no es un múltiplo del tamaño del vector grande, sacaría un warning avisándonos de ello pero realizaría la operación rellenando el vector pequeño hasta la posición necesaria.

```
v1 = c(4, 6, 8, 24)
v2 = c(10, 2, 4)
v1 + v2
```

Warning in v1 + v2: longitud de objeto mayor no es múltiplo de la longitud de uno menor

```
[1] 14  8 12 34
```

5.2. Factores.

Los factores son la forma que tiene R de manejar datos categóricos. Los factores tienen niveles que son los posibles valores que pueden tomar.

Veamos un ejemplo de definición de factor a partir de un vector.

```
cats = c(rep("malo", 2), rep("regular", 2), rep("bueno", 3) )
# Transformación de un vector en un factor:
fcats = factor(cats)
fcats
```

```
[1] malo    malo    regular regular bueno    bueno    bueno
Levels: bueno malo regular
```

5. Tipos de datos

```
# Código numérico para cada categoría
as.numeric(fcats)
```

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

```
# Niveles de categorías reconocidos
levels(fcats)
```

```
[1] "bueno"    "malo"     "regular"
```

Otros ejemplos para ver como podemos trabajar con factores:

```
fgenero = factor(c("m", "m", "m", "m", "m"), levels = c("f","m"))
fgenero
```

```
[1] m m m m m
Levels: f m
```

```
# Con una variable de tipo factor podemos contar las ocurrencias de cada pos.
table(fgenero)
```

```
fgenero
f m
0 5
```

```
a = factor(c("adult", "adult", "juvenil", "juvenil", "adult"))
# Se pueden cruzar dos factores
t = table(a,fgenero)
t
```


5.2. Factores.

```
      fgenero
a      f m
adult  0 3
juvenil 0 2
```

```
# Cálculo de las frecuencias marginales de cada uno de los factores
margin.table(t, 1)
```

```
a
  adult juvenil
      3       2
```

```
margin.table(t, 2)
```

```
fgenero
f m
0 5
```

La función `gl()` puede utilizarse para generar secuencias a partir de factores. Esta función recibe dos parámetros, el primer indica el número de niveles y el segundo indica el número de repeticiones de cada nivel.

```
gl(3, 5)
```

```
[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
Levels: 1 2 3
```

```
gl(2, 5, labels = c("female", "male"))
```

```
[1] female female female female female male  male  male  male  male
Levels: female male
```

5. Tipos de datos

5.3. Matrices y Arrays.

Los arrays nos permiten almacenar elementos en diversas dimensiones. Las matrices son un caso especial de arrays con dos dimensiones. Tanto los arrays como las matrices en R no son más que vectores con un atributo particular que es la dimensión. Por lo tanto, también son estructuras **homogéneas** de almacenamiento.

5.3.1. Construcción de arrays a partir de vectores.

Por lo comentado anteriormente podemos construir un array sin más que proporcionándole al vector donde están almacenados los datos con unas dimensiones.

```
m = c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23)
m
```

```
[1] 45 23 66 77 33 44 56 12 78 23
```

```
# El contenido del vector se reparte a través de la matriz con 2 filas y 5 c
dim(m) = c(2, 5)
m
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,]   45   66   33   56   78
[2,]   23   77   44   12   23
```

5.3.2. Función `matrix()`.

También se puede definir directamente una matriz haciendo uso de la función `matrix()`:

```
# 1º argumento: contenido de la matriz en forma de vector
# 2º argumento: número de filas
# 3º argumento: número de columnas
# De esta forma, la matriz se va rellenoando por columnas
m = matrix(c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23), 2, 5)
m
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,]   45   66   33   56   78
[2,]   23   77   44   12   23
```

```
# Si queremos que la matriz se vaya rellenoando por filas, debemos utilizar el argumento byrow
m = matrix(c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23), 2, 5, byrow = TRUE)
m
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,]   45   23   66   77   33
[2,]   44   56   12   78   23
```

5.3.3. Función `array()`.

Para facilitar la creación de arrays tenemos la función `array()`, que funciona de un modo similar a `matrix()`. En el primer argumento se le indica el contenido del array y mediante el argumento `dim` le podemos especificar el tamaño de cada una de las dimensiones.

5. Tipos de datos

```
# Array de dimensión 4x3x2
a = array(5:29, dim=c(4, 3, 2))
a
```

, , 1

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

, , 2

	[,1]	[,2]	[,3]
[1,]	17	21	25
[2,]	18	22	26
[3,]	19	23	27
[4,]	20	24	28

5.3.4. Nombres de filas y columnas.

En R tenemos las funciones `colnames()` y `rownames()` que nos permiten dar nombres a las filas y a las columnas de una matriz.

```
results = matrix(c(10, 20, 30, 40, 21, 32, 43, 13), 2, 4, byrow = TRUE)
colnames(results) = c("1qrt", "2qrt", "3qrt", "4qrt")
rownames(results) = c("row1", "row2")
results
```

	1qrt	2qrt	3qrt	4qrt
row1	10	20	30	40
row2	21	32	43	13

5.3.5. Indexación de elementos.

Del mismo modo que podíamos acceder a los elementos de un vector podemos acceder a los elementos de un array, la única diferencia es que en este caso hay que indicar tantos índices como dimensión tenga el array.

```
# EJEMPLOS CON MATRICES:
```

```
# Todos los elementos de una fila
```

```
m[1, ]
```

```
[1] 45 23 66 77 33
```

```
# Todos los elementos de una columna
```

```
m[, 4]
```

```
[1] 77 78
```

```
# Elementos de una fila menos los que se encuentran en ciertas columnas
```

```
m[1, -c(3, 5)]
```

```
[1] 45 23 77
```

```
# Elementos de una columna menos los que se encuentran en una determinada fila
```

```
m[-2, 1]
```

```
[1] 45
```

```
# EJEMPLOS CON ARRAYS:
```

```
a[1, 3, 2]
```

```
[1] 25
```

5. Tipos de datos

```
a[1, , 2]
```

```
[1] 17 21 25
```

```
a[4, 3, ]
```

```
[1] 16 28
```

```
# Nos quedamos con los elementos que ocupan las filas 2 y 3, todas las columnas
# solo los que se encuentran en la primera tabla
a[c(2, 3), , -2]
```

```
      [,1] [,2] [,3]
[1,]     6    10    14
[2,]     7    11    15
```

Como consecuencia de la indexación en una matriz podemos obtener como resultado un vector. Si queremos que el resultado siga siendo una matriz, incluso si es una matriz formada por una sola fila o columna, podemos usar la siguiente expresión:

```
m[1, , drop = FALSE]
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,]   45   23   66   77   33
```

```
m[, 4, drop = FALSE]
```

```
      [,1]
[1,]    77
[2,]    78
```

5.3. Matrices y Arrays.

Como ya se ha visto en el apartado anterior, es posible darle nombres a las filas y las columnas de una matriz. Por lo que, la indexación de elementos también se puede realizar mediante estos nombres.

```
results["row1", ]
```

```
1qrt 2qrt 3qrt 4qrt
  10   20   30   40
```

```
results["row2", c("1qrt", "4qrt")]
```

```
1qrt 4qrt
  21   13
```

5.3.6. Unión de matrices y vectores

Las funciones `cbind()` y `rbind()` nos permiten construir arrays mediante la unión (por columnas o por filas, respectivamente) de vectores o matrices.

```
m1 = matrix(c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23), 2, 5)
m1
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,]   45   66   33   56   78
[2,]   23   77   44   12   23
```

```
# Construimos una nueva matriz a partir de la unión del vector (4 ,76) y
# la 4ª columna de m1
m2 = cbind(c(4, 76), m1[, 4])
m2
```

5. Tipos de datos

```
      [,1] [,2]
[1,]    4  56
[2,]   76  12
```

```
m3 = matrix(rep(12, 20), 4, 5)
m3
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,]   12   12   12   12   12
[2,]   12   12   12   12   12
[3,]   12   12   12   12   12
[4,]   12   12   12   12   12
```

```
# Construimos una nueva matriz a partir de la unión de la fila 1 de m1
# y la fila 3 de m3
m4 = rbind(m1[1, ], m3[3, ])
m4
```

```
      [,1] [,2] [,3] [,4] [,5]
[1,]   45   66   33   56   78
[2,]   12   12   12   12   12
```

5.3.7. Operaciones aritméticas

Tanto a las matrices como a los arrays se les aplican las mismas reglas aritméticas que a los vectores a la hora de realizar operaciones aritméticas con ellas.

```
m = matrix(c(45, 23, 66, 77, 33, 44, 56, 12, 78, 23), 2, 5)
m
```


5.3. Matrices y Arrays.

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	45	66	33	56	78
[2,]	23	77	44	12	23

```
m * 3
```

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	135	198	99	168	234
[2,]	69	231	132	36	69

```
m1 = matrix(c(45, 23, 66, 77, 33, 44), 2, 3)
m1
```

	[,1]	[,2]	[,3]
[1,]	45	66	33
[2,]	23	77	44

```
m2 = matrix(c(12, 65, 32, 7, 4, 78), 2, 3)
m2
```

	[,1]	[,2]	[,3]
[1,]	12	32	4
[2,]	65	7	78

```
m1 + m2
```

	[,1]	[,2]	[,3]
[1,]	57	98	37
[2,]	88	84	122

5. Tipos de datos

5.4. Listas.

Una lista en R consiste en una colección ordeanda de elemntos. Los elementos u objetos almacenados en ella se conocen con el nombre de componentes. A diferencia de los vectores o arrays, los componentes de una lista pueden ser de diferente tipo. A estos componentes les podemos dar nombres igual que hacíamos en los arrays.

5.4.1. Creación de una lista.

Para crear una lista no hay más que darle un nombre a la lista e indicarle a la función `list()` cuales van a ser los elementos que vamos a guardar en ella. A cada uno de estos elementos le podemos dar un nombre.

```
# Lista de tres componentes que llamamos stud.id, stud.name y stud.marks
my.lst = list(stud.id=69449, stud.name="Ana", stud.marks=c(5.25, 10, 7.5, 9.25))
```

No es obligatorio darle nombre a los componentes de una lista cuando la creamos. También existe la opción de darles nombre después de haber creado la lista.

```
my.lst2 = list(12584, "Marcos", c(8.5, 10, 17, 9.75))
my.lst2
```

```
[[1]]
```

```
[1] 12584
```

```
[[2]]
```

```
[1] "Marcos"
```

```
[[3]]
```

```
[1] 8.50 10.00 17.00 9.75
```

```
names(my.lst2) = c("id", "name", "marks")
my.lst2
```

```
$id
[1] 12584
```

```
$name
[1] "Marcos"
```

```
$marks
[1] 8.50 10.00 17.00 9.75
```

La función **unlist()** nos devuelve un vector con tantos elementos como elementos hubiera en la lista que se le pasa por parámetro. Como los vectores solo pueden estar compuestos por elementos del mismo tipo, el uso de esta función además provocará cambios en los tipos de elementos. Lo más normal es que transforme todos los elementos a cadenas de caracteres.

```
unlist(my.lst2)
```

id	name	marks1	marks2	marks3	marks4
"12584"	"Marcos"	"8.5"	"10"	"17"	"9.75"

5.4.2. Indexación de elementos.

Si a la hora de crear nuestra lista le hemos dado nombre a los componentes, podemos acceder a ellos a través de su nombre. Si no le hemos dado nombre, siempre podemos acceder a ellos a través del índice de la posición que ocupan.

5. Tipos de datos

```
my.lst$stud.id
```

```
[1] 69449
```

```
my.lst$stud.name
```

```
[1] "Ana"
```

```
# Accedemos al valor del componente que se encuentra en la 1ª posición  
my.lst[[1]]
```

```
[1] 69449
```

```
# Accedemos al valor del componente que se encuentra en la 3ª posición  
my.lst[[3]]
```

```
[1] 5.25 10.00 7.50 9.25
```

Si realizamos la indexación por índice de posición y solo utilizamos un corchete, el resultado que obtenemos es una sublista de la lista original con el componente deseado.

```
my.lst[1]
```

```
$stud.id  
[1] 69449
```

5.4. Listas.

```
# Comprobamos que el tipo de dato es diferente  
mode(my.lst[1])
```

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

```
mode(my.lst[[1]])
```

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

5.4.3. Añadir y quitar elementos a una lista.

Dada una lista siempre podemos añadir y quitar elementos de ella. Veamos un ejemplo:

```
my.lst$parents.names = c("Ana", "Miguel")  
my.lst
```

```
$stud.id  
[1] 69449
```

```
$stud.name  
[1] "Ana"
```

```
$stud.marks  
[1] 5.25 10.00 7.50 9.25
```

```
$parents.names  
[1] "Ana" "Miguel"
```

5. Tipos de datos

```
my.lst = my.lst[-5]  
my.lst
```

```
$stud.id  
[1] 69449
```

```
$stud.name  
[1] "Ana"
```

```
$stud.marks  
[1] 5.25 10.00 7.50 9.25
```

```
$parents.names  
[1] "Ana" "Miguel"
```

Otra opción de añadir elementos a una lista es concatenando varias listas.

```
other = list(age = 19, sex = "male")  
lst = c(my.lst, other)  
lst
```

```
$stud.id  
[1] 69449
```

```
$stud.name  
[1] "Ana"
```

```
$stud.marks  
[1] 5.25 10.00 7.50 9.25
```

```
$parents.names
```

5.5. Data Frames.

```
[1] "Ana"      "Miguel"
```

```
$age
```

```
[1] 19
```

```
$sex
```

```
[1] "male"
```

Mediante la función **length()** podemos saber el número de componentes de una lista en todo momento.

```
length(my.lst)
```

```
[1] 4
```

5.5. Data Frames.

- Los **datos estructurados** son aquellos que podemos organizar aplicando un esquema previo (estructura), que es conocido antes de realizar su análisis y se mantiene fijo.
 - Ejemplo: una tabla de una base de datos relacional, un archivo CSV.
- Los **datos no estructurados** son aquellos que no podemos organizar a priori, puesto que pueden contener cualquier tipo de información en cualquier formato. Así pues, primero debemos analizar dicho contenido para extraer información valiosa.
 - Ejemplo: transcripciones de conversaciones sobre las que aplicamos PLN.

5. Tipos de datos

El Data Frame es la estructura de datos recomendada para almacenar tablas de datos en R. Desde el punto de vista estructural son similares a las matrices, ya que se trata de estructuras de datos bi-dimensionales. Sin embargo, al contrario que las matrices, e imitando el comportamiento de las listas, los datos que se almacenan en un Data Frame pueden ser de diferentes tipos. En este sentido pueden verse como una extensión de las listas. En definitiva, se trata de **estructuras de almacenamiento bi-dimensionales de datos heterogéneos**.

5.5.1. Creación de un Data Frame.

Para la creación de un Data Frame en R tenemos la función **data.frame()**. La forma de definir el Data Frame es equivalente a la lista, indicamos el nombre de cada columna y los valores que se van almacenar en ella.

Es conveniente identificar cada columna con una variable y cada fila con una observación (compuesta por el conjunto de valores de cada una de las variables).

```
my.dataset = data.frame(site=c('A','B','A','A','B'),
                        estacion=c('Inverno','Verano','Verano','Primavera','Otoño'),
                        pH = c(7.4, 6.3, 8.6, 7.2, 8.9))
my.dataset
```

	site	estacion	pH
1	A	Inverno	7.4
2	B	Verano	6.3
3	A	Verano	8.6
4	A	Primavera	7.2
5	B	Otoño	8.9

5.5. Data Frames.

Normalmente crearemos un Data Frame a partir de la lectura de los datos de un fichero. En el siguiente capítulo veremos como se realiza esta operación.

5.5.2. Características de un Data Frame.

Podemos conocer el número de filas, columnas y la dimensión de un Data Frame mediante las funciones **nrow()**, **ncol()** y **dim()**.

```
nrow(my.dataset)
```

```
[1] 5
```

```
ncol(my.dataset)
```

```
[1] 3
```

```
dim(my.dataset)
```

```
[1] 5 3
```

También podemos conocer o cambiar el nombre de las columnas de un Data Frame mediante la función **names()**.

```
names(my.dataset)
```

```
[1] "site"      "estacion" "pH"
```

5. Tipos de datos

```
names(my.dataset)=c("area","season","pH")
my.dataset
```

	area	season	pH
1	A	Inverno	7.4
2	B	Verano	6.3
3	A	Verano	8.6
4	A	Primavera	7.2
5	B	Otoño	8.9

5.5.3. Indexación de elementos.

Podemos acceder a los elementos de un Data Frame del mismo modo que accedemos a los elementos de una matriz, tanto por su posición como por el nombre de la columna.

```
my.dataset[3, 2]
```

```
[1] "Verano"
```

```
# Acceso a una columna determina mediante su nombre:
my.dataset$pH
```

```
[1] 7.4 6.3 8.6 7.2 8.9
```

```
my.dataset[["site"]]
```

```
NULL
```

5.5. Data Frames.

Aprovechando la capacidad de R para obtener sub-secciones, esta opción también es posible con un Data Frame. Es decir, a partir de un Data Frame podemos obtener otro Data Frame que consista solo en una parte del primero. Esta operación se realiza mediante la aplicación de una máscara, que no es más que una expresión lógica.

```
# Construimos un nuevo Data Frame con las filas que tengan pH mayores que 7:  
my.dataset[my.dataset$pH>7, ]
```

	area	season	pH
1	A	Inverno	7.4
3	A	Verano	8.6
4	A	Primavera	7.2
5	B	Otoño	8.9

```
# El nuevo Data Frame está formado solo por los valores de pH de aquellas filas que tienen  
# site=A:  
my.dataset[my.dataset$site == "A", "pH"]
```

```
numeric(0)
```

```
# Nos quedamos con las filas que tienen season=Summer y las columnas site y pH:  
my.dataset[my.dataset$season == "Summer", c("area","pH")]
```

```
[1] area pH  
<0 rows> (o 0- extensión row.names)
```

Otra opción de obtener una sub-sección de un Data Frame es usando la función `subset()`.

5. Tipos de datos

```
subset(my.dataset, pH >8)
```

```
  area season  pH  
3    A Verano 8.6  
5    B Otoño 8.9
```

```
subset(my.dataset, season == "Summer", season:pH)
```

```
[1] season pH  
<0 rows> (o 0- extensión row.names)
```

La diferencia con el procedimiento anterior es que, si queremos asignar nuevos valores a una sub-sección de un Data Frame, no podemos hacerlo con la función **subset()**, tenemos que obtener la sub-sección con la aplicación de una máscara.

```
my.dataset[my.dataset$season == "Summer", "pH"] =  
  my.dataset[my.dataset$season == "Summer", "pH"]+1
```

5.5.4. Añadir nuevas columnas.

Se pueden añadir nuevas columnas a un Data Frame del mismo modo que hacíamos con las listas. La única restricción que debemos tener en cuenta es que la nueva columna debe tener el mismo número de filas que las anteriores.

```
my.dataset$N03 = c(234.5, 256.6, 654.1, 356.7, 776.4)  
my.dataset
```

	area	season	pH	N03
1	A	Inverno	7.4	234.5
2	B	Verano	6.3	256.6
3	A	Verano	8.6	654.1
4	A	Primavera	7.2	356.7
5	B	Otoño	8.9	776.4

5.5.5. Extensión de Data Frames: tibble.

Los paquetes **tibble** (Wickham et al., 2016) y **dplyr** (Wickham and Fancis, 2015) nos proporcionan un conjunto de operaciones muy útil para facilitar la manipulación de datos. El paquete **tibble** define la estructura de datos denominada *tibbles* considerada como un caso especial de Data Frame. Los *tibbles* modifican algunos de los comportamientos estandar que hemos visto de los Data Frame:

- los *tibbles* nunca cambian las columnas caracter en factores como hacen los Data Frame por defecto
- los *tibbles* tienen más flexibilidad a la hora de nombrar a las columnas.
- los métodos de representación que tienen los *tibbles* son más convenientes que los de los Data Frame cuando se trabaja con grandes cantidades de datos.

Para crear un *tibble* utilizamos la función **tibble()** de R. Su funcionamiento es muy similar al de la función **data.frame()**. Como se puede observar en el siguiente ejemplo, los valores de cada columna se calculan secuencialmente permitiendonos usar los valores de la columna anterior, y los vectores de caracteres no se convierten en factores.

```
library(tibble)
dat = tibble(TempCels = sample(-10:40, size=100, replace=TRUE),
              TempFahr = TempCels*9/5+32,
```

5. Tipos de datos

```
Location =rep(letters[1:2], each=50))  
dat
```

```
# A tibble: 100 x 3  
  TempCels TempFahr Location  
    <int>    <dbl> <chr>  
1         3     37.4 a  
2        -3     26.6 a  
3        19     66.2 a  
4        -1     30.2 a  
5        18     64.4 a  
6         6     42.8 a  
7        -7     19.4 a  
8         2     35.6 a  
9        32     89.6 a  
10       39    102. a  
# i 90 more rows
```

Cualquier Data Frame estandar se puede convertir en un *tibble*. Como ejemplo para visualizar este comportamiento vamos a usar el famoso dataset *Iris*, está disponible directamente en R y contiene 150 filas con 5 variables.

```
# Cargamos el dataset Iris en un Data Frame estandar  
data(iris)  
dim(iris)
```

```
[1] 150  5
```

```
class(iris)
```

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

5.5. Data Frames.

```
library(tibble)
# Convertimos el Data Frame a tibble
ir = as_tibble(iris)
# No hemos creado una nueva copia del Data Frame. El nuevo objeto es todavía de la clase
# data.frame pero también pertenece a las clases tbl_df (la clase de los tibble) y tbl
# (es una generalización de la anterior).
class(ir)
```

```
[1] "tbl_df"      "tbl"        "data.frame"
```

```
ir
```

```
# A tibble: 150 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
      <dbl>       <dbl>       <dbl>       <dbl> <fct>
1         5.1         3.5         1.4         0.2 setosa
2         4.9         3         1.4         0.2 setosa
3         4.7         3.2         1.3         0.2 setosa
4         4.6         3.1         1.5         0.2 setosa
5          5          3.6         1.4         0.2 setosa
6         5.4         3.9         1.7         0.4 setosa
7         4.6         3.4         1.4         0.3 setosa
8          5          3.4         1.5         0.2 setosa
9         4.4         2.9         1.4         0.2 setosa
10        4.9         3.1         1.5         0.1 setosa
# i 140 more rows
```

Otra diferencia fundamental entre los *tibbles* y los Data Frame es la indexación de elementos. Después de aplicar una máscara a un Data Frame hemos visto que lo que obteníamos era un vector. Sin embargo, con los *tibbles* esto no pasa, después de aplicar una máscara lo que obtenemos es otro *tibble*. Veamos esta diferencia de comportamiento en el siguiente ejemplo:

5. Tipos de datos

```
iris[1:15, "Petal.Length"]
```

```
[1] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 1.5 1.6 1.4 1.1 1.2
```

```
class(iris[1:15, "Petal.Length"])
```

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

```
# Con drop=FALSE podemos eliminar este comportamiento de los Data Frame  
iris[1:15, "Petal.Length",drop=FALSE]
```

	Petal.Length
1	1.4
2	1.4
3	1.3
4	1.5
5	1.4
6	1.7
7	1.4
8	1.5
9	1.4
10	1.5
11	1.5
12	1.6
13	1.4
14	1.1
15	1.2

```
class(iris[1:15, "Petal.Length",drop=FALSE])
```

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


5.5. Data Frames.

```
ir[1:15, "Petal.Length"]
```

```
# A tibble: 15 x 1
  Petal.Length
      <dbl>
1         1.4
2         1.4
3         1.3
4         1.5
5         1.4
6         1.7
7         1.4
8         1.5
9         1.4
10        1.5
11        1.5
12        1.6
13        1.4
14        1.1
15        1.2
```

El paquete **dplyr** nos proporciona las funciones **select()** y **filter()** para hacer indexación de elementos. La función **select()** se usa para seleccionar un conjunto de columnas y la función **filter()** para seleccionar un conjunto de filas.

```
library(dplyr)
```

Adjuntando el paquete: 'dplyr'

5. Tipos de datos

The following objects are masked from 'package:stats':

filter, lag

The following objects are masked from 'package:base':

intersect, setdiff, setequal, union

```
# Nos quedamos con las filas de la especie "setosa", de la cual solo queremos  
# que se llaman Petal.Width y Petal.Length  
select(filter(ir, Species=="setosa"), Petal.Width, Petal.Length)
```

```
# A tibble: 50 x 2  
  Petal.Width Petal.Length  
    <dbl>         <dbl>  
1         0.2         1.4  
2         0.2         1.4  
3         0.2         1.3  
4         0.2         1.5  
5         0.2         1.4  
6         0.4         1.7  
7         0.3         1.4  
8         0.2         1.5  
9         0.2         1.4  
10        0.1         1.5  
# i 40 more rows
```

```
# Podemos hacer lo mismo utilizando el operador pipe:  
filter(ir, Species=="setosa") %>% select(Petal.Width, Petal.Length)
```

```
# A tibble: 50 x 2  
  Petal.Width Petal.Length
```

5.5. Data Frames.

	<dbl>	<dbl>
1	0.2	1.4
2	0.2	1.4
3	0.2	1.3
4	0.2	1.5
5	0.2	1.4
6	0.4	1.7
7	0.3	1.4
8	0.2	1.5
9	0.2	1.4
10	0.1	1.5

i 40 more rows

NOTA: el operador pipe (`%>%`) se puede aplicar a cualquier función de R. Significa que, lo que está a su izquierda se pasa como el primer argumento a la función que está a su derecha. Es decir, una expresión del tipo `x %>% f(y)` es equivalente a `f(x,y)`.

6. Tidy data

En este capítulo se presenta una introducción a tidyverse. Se trata de un conjunto de paquetes R organizados en torno a unos principios de organización de datos para facilitar su análisis, presentados por Hadley Wickham en 2014. A los primeros paquetes publicados por el propio Wickham para implementar estos principios (como `dplyr` o `tidyr`), pronto se unieron muchos otros, creando un verdadero ecosistema propio dentro del catálogo de paquetes de R.

Aunque en sus orígenes, de manera informal, se popularizó el término “*hadley-verse*” para denominar a estos paquetes, pronto el termino más neutro “*tidyverse*” tomo su lugar.

Actualmente, los múltiples paquetes que conforman el tidyverse han contribuido a unificar la forma de programar y trabajar con datos en el lenguaje R, introduciendo muchas funcionalidades y, sobre todo, marcando un estilo de trabajo y organización de los datos consistente en todos sus componentes.

En las siguientes secciones nos concentraremos en los dos paquetes básicos de tidyverse: `dplyr` y `tidyr`. Estos dos paquetes junto con el operador `%>%` (pipe) del paquete `magrittr` forman el núcleo básico de trabajo del tidyverse.

Para cargar el tidyverse en nuestra sesión ejecutamos:

```
library(tidyverse)
```

6. Tidy data

6.1. Principios: “tidy data”

En septiembre de 2014, la revista Journal of Statistical Software publicó un artículo de H. Wickham (Wickham, 2014) en el que presentaba algunos principios básicos para estructurar la organización de conjuntos de datos. Su objetivo era facilitar la preparación, modelado y visualización de datos en todas las fases de un proyecto.

Primero establecemos un vocabulario de términos estándar:

- Un conjunto de datos es una colección de **valores**, que pueden ser cuantitativos (numéricos) o cualitativos (categóricos).
- Una **variable** representa el conjunto de todos los valores que miden el mismo atributo (altura, temperatura, duración, etc.).
- Una **observación** contiene todos los valores medidos sobre la misma unidad de análisis (una persona, un día, una provincia, etc.) para todos los atributos.

Según este artículo, los tres **principios básicos** para que un conjunto de datos se organice de acuerdo a las normas de **tidy data** son (Wickham, 2014):

1. Cada **variable** está en **una columna**.
2. Cada **observación** está en **una fila**.
3. Cada **tipo de unidad de observación** forma **una tabla**.

Por “tipo de unidad de observación” podemos entender una persona, un tipo de objeto o cualquier otro tipo de elemento que constituya un *tipo de unidad de análisis* en nuestro estudio. Cualquier conjunto de datos que no siga estos principios se considera *messy data*.

Estos principios no son en absoluto originales de Wickham, sino que el autor se limitó a reexpresar algunos principios de normalización del álgebra relacional de Codd (Codd, 1990), aunque en un lenguaje más accesible

6.1. Principios: “tidy data”

para estadísticos y otros profesionales del análisis de datos que no tengan, necesariamente, conocimientos de teoría de bases de datos.

Podemos notar rápidamente la influencia de estos principios en un ejemplo sencillo. Consideremos la siguiente tabla, presentada como ejemplo en el artículo original de Wickham, con datos de un hipotético análisis clínico sobre pacientes:

Tabla 6.1.: (#tab:messy-data-1) Ejemplo de un conjunto de datos típico.

	treatment_a	treatment_b
John Smith	—	2
Jane Doe	16	11
Mary Johnson	3	1

Si tenemos alguna experiencia previa analizando datos con herramientas como hojas de cálculo, y reflexionamos un poco, es muy probable que hayamos visto con anterioridad una estructura de organización de datos parecida. Ahora, presentamos una segunda versión, con los mismos datos pero organizados de manera distinta:

Tabla 6.2.: (#tab:messy-data-2) El mismo dataset que en la Tabla @ref(tab:messy-data-1), pero con estructura diferente.

	John Smith	Jane Doe	Mary Johnson
treatment_a	—	16	3
treatment_b	2	11	1

Si comparamos la Tabla @ref(tab:messy-data-1) y la Tabla @ref(tab:messy-data-2) podemos comprobar que contienen la misma información. Sin embargo, la estructura es diferente. En la Tabla @ref(tab:messy-data-1) la primera columna contiene los nombres de los pacientes, mientras que las

6. Tidy data

dos siguientes hacia la derecha contienen los resultados de la prueba de cada paciente para el tratamiento A y el B, respectivamente. En su lugar, en la Tabla @ref(tab:messy-data-2) las filas recogen los datos de todos los pacientes para cada tratamiento, y las columnas corresponden a los datos de cada paciente.

Aunque en ambos casos los datos son “correctos” en sentido técnico (la información es la misma en ambos casos y suponemos que no hay fallos en la introducción de datos), enseguida podemos imaginar que un script de R (o cualquier otro lenguaje) que espere recibir los datos en el formato de la Tabla @ref(tab:messy-data-1) y, en cambio, los reciba en el formato de la Tabla @ref(tab:messy-data-2) seguramente va a devolver errores. Y ello pese a que la información es correcta en ambos casos.

En consecuencia, es importante establecer explícitamente un acuerdo entre los investigadores para organizar siempre los datos de la misma manera. De esta forma, sabemos qué formato de datos esperamos encontrar y se puede intentar “reclamar” al origen de los datos que siga esta convención de representación de la información (aunque no siempre esto es posible).

Tabla 6.3.: (#tab:tidy-data) El mismo dataset, pero ahora en formato *tidy data*.

name	treatment	result
John Smith	a	—
Jane Doe	a	16
Mary Johnson	a	3
John Smith	b	2
Jane Doe	b	11
Mary Johnson	b	1

La tabla @ref(tab:tidy-data) muestra cómo se representan los datos siguiendo la convención establecida por las reglas de tidy data:

6.2. El paquete tidyverse

- Hay 3 variables (nombre de paciente, tratamiento y resultado), una en cada columna.
- Hay 6 filas, una por cada unidad de observación (aquí la unidad de observación es la combinación paciente + tratamiento).
- En la intersección de cada fila y cada columna hay un solo dato.

Siempre que haya variables *categorías* involucradas, el formato tidy data obliga a poner una fila nueva por cada persona y nivel registrado de cada variable categórica. Ciertamente, esto obliga a repetir información en varias filas (por ejemplo, nombre de tratamiento), y puede parecer que no es muy eficiente computacionalmente. No obstante, como veremos después R guarda los niveles de las variables categóricas internamente como números enteros, y solo cuando tiene que mostrar esa información en pantalla “traduce” esos códigos a la etiqueta de identificación que le hemos asignado a cada nivel. Así que no hay tanta pérdida de eficiencia como podría parecer a primera vista.

6.2. El paquete tidyverse

El sitio web <http://www.tidyverse.org> centraliza gran parte de la información general sobre la filosofía de trabajo y los paquetes principales que componen el ecosistema del tidyverse. En particular en la URL <https://www.tidyverse.org/packages/> encontramos una página que resume la lista de principales paquetes que podemos encontrar.

Aunque en este capítulo nos centraremos sobre todo en `dplyr` y `tidyr`, existen muchos otros paquetes que proporcionan funcionalidades extra muy interesantes, tanto para trabajo con tipos de datos específicos (`forcats`, `lubridate`, `hms`) para operaciones de comunicación (`httr`, `rvest`), o para estructurar y unificar informes sobre modelos y anotación de datasets con resultados (`broom`). En la siguiente sección se muestra una tabla con los principales paquetes que, en este momento, son utilizados con frecuencia en el tidyverse.

6. Tidy data

Mención especial merece un importante esfuerzo que se está realizando en la actualidad para modernizar el proceso de diseño, implementación, ajuste y mantenimiento de modelos, siguiendo los principios de tidy data y completamente integrado con el resto de elementos del tidyverse. Aunque H. Wickham inició este trabajo con el paquete `modelr` (Wickham & Grommund, 2017), pronto fue evidente que se necesitaría mucho más esfuerzo para llevar la iniciativa a buen puerto. En los últimos años, RStudio (la empresa en la que Wickham trabaja) ha contratado a algunos de los más destacados desarrolladores de la comunidad R en todo el mundo (e.g. Max Kuhn, Jenny Brian, etc.) para acelerar el progreso en este frente.

Kuhn, principal autor del paquete `caret` (Kuhn & Johnson, 2013), uno de los más usados para diseño e implementación de modelos de aprendizaje máquina en R, se encarga ahora de liderar el desarrollo de `tidymodels`, un metapaquete similar a `tidyverse` que automatiza la instalación y carga de un **conjunto de paquetes** (algunos existentes en tidyverse y otros nuevos) para **desarrollo y evaluación de modelos**. Por el momento, aunque conviene no perder de vista esta iniciativa el proyecto sigue todavía lejos de alcanzar una versión estable y consolidada.

6.2.1. Resumen de paquetes principales en tidyverse

La Tabla @ref(tab:tidyverse-packages) resume algunos de los paquetes más importantes que encontraremos en el ecosistema tidyverse.

Tabla	
Paquete	Descripción
‘tibble’	Tipo de datos básico para almacenar tablas de datos, similar a un data frame
‘dplyr’	Proporciona una gramática para manipulación de datos, inspirada en las operaciones de SQL
‘tidyr’	Herramientas que ayudan a transformar los datos para que cumplan las reglas de tidy data
‘ggplot2’	Sistema que proporciona un lenguaje declarativo para crear visualizaciones de datos
‘readr’	Funciones eficientes para lectura/escritura de datos estructurados, que proporcionan una interfaz simple y rápida

6.2. El paquete tidyverse

‘purrr’	Conjunto de herramientas para programación funcional con R, que permite reemplazar bucles
‘stringr’	Conjunto de funciones para operativizar el trabajo con cadenas de caracteres en el contexto de
‘forcats’	Conjunto de funciones para trabajo con variables categóricas (llamadas <i>*factores*</i> en R.

6.2.2. Método de trabajo con tidyverse

El método de trabajo en tidyverse suele estar fuertemente ligado a la utilización del operador *pipe* (`%>%`) del paquete `magrittr`. Esencialmente, este operador simplifica la sintaxis necesaria para realizar llamadas anidadas a funciones, de forma que el resultado de la operación anterior actúe como entrada del siguiente paso. Imaginemos que tenemos un objeto similar a un data frame (en la siguiente sección se introducen los objetos tibble) llamado `df`, al que queremos aplicar dos funciones de preparación de datos:

```
f2(f1(df), arg_f2) # Se complica rápido con muchas funciones
                  # y es difícil de leer. La función f2 toma
                  # un argumento adicional para definir la operación
                  # a realizar
```

Por el contrario, si usamos el operador `%>%` podemos escribir cada paso de la cadena de operaciones en una línea diferente. De esta manera, podemos sintácticamente separar cada paso de nuestra cadena de operaciones, de forma que el código queda mucho más legible:

```
# x %>% f(y) se convierte en f(x, y)
df %>%
  f1 %>%      # Recibe como primer argumento de entrada df
  f2(arg_f2) # Recibe el resultado del paso anterior, f1(df)
              # arg_f2 se pasa a continuación como argumento de entrada
```

6. Tidy data

6.2.3. Objetos tibble

El paquete `tibble` de `tidyverse` proporciona un tipo de dato especial para representar de forma más eficiente tablas de datos, siguiendo una estructura similar a un data frame pero con características mucho más eficientes.

Veamos un ejemplo:

```
head(iris) # data frame estándar
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa

```
iris_tibble <- as_tibble(iris)
print(iris_tibble)
```

```
# A tibble: 150 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
    <dbl>         <dbl>         <dbl>         <dbl> <fct>
1     5.1         3.5         1.4         0.2 setosa
2     4.9         3         1.4         0.2 setosa
3     4.7         3.2         1.3         0.2 setosa
4     4.6         3.1         1.5         0.2 setosa
5     5          3.6         1.4         0.2 setosa
6     5.4         3.9         1.7         0.4 setosa
7     4.6         3.4         1.4         0.3 setosa
8     5          3.4         1.5         0.2 setosa
9     4.4         2.9         1.4         0.2 setosa
```

6.2. El paquete tidyverse

```
10          4.9          3.1          1.5          0.1 setosa
# i 140 more rows
```

Como vemos, un objeto de este tipo es muy similar a un data frame, pero añade explícitamente en la salida información sobre el tipo de dato de cada columna. Un aspecto importante que resaltaremos más adelante es que, por defecto, tibble intenta asignar el tipo de dato adecuado a cada columna, excepto en un caso: si la columna contiene cadenas de caracteres como valores entonces **no los transforma automáticamente a variable categórica en R** (las variables categóricas en R, recordemos, se llaman factores). En su lugar, deja los valores de la columna como objetos string (en la cabecera se lee tipo `<str>`). Si queremos que se conviertan a tipo de dato categórico lo tendremos que indicar explícitamente (veremos cómo más adelante). Comentamos este aspecto porque, en el ejemplo anterior, vemos que la columna del tipo de flor sí se ha identificado como de tipo factor (`<fct>`), pero esto se debe a que originalmente la columna `iris$Species` ya era de tipo factor:

```
class(iris$Species)
```

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

Esto no es la norma habitual cuando leemos datos de fuentes externas, por ejemplo de un fichero, una API o una base de datos, por lo que conviene recordar esta característica para no arrastrar el error en pasos posteriores.

Además, la función `str(...)` permite obtener información adicional muy útil sobre el formato de las columnas de un objeto tibble:

```
str(iris_tibble)
```

6. Tidy data

```
tibble [150 x 5] (S3: tbl_df/tbl/data.frame)
 $ Sepal.Length: num [1:150] 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num [1:150] 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num [1:150] 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num [1:150] 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...
```

Mencionar de pasada que existe también `data.table`, competidor de este paquete (y con muy buen rendimiento...)

6.3. Preparación de datos con tidyverse: tidyr

Puesto que tidyverse asume que los datos se organizarán en tablas que siguen estrictamente sus principios de organización, el primer paso es asegurarnos de que, en efecto, los datos de entrada cumplen estos requisitos.

En caso contrario, el paquete `tidyr` proporciona una serie de herramientas que nos van a facilitar el trabajo de convertir nuestros datos a este formato. Algunas de ellas se resumen en la Tabla @ref(tab:tidyr-funcs).

Funciones	Descripción
Pivoting	Operación: Cambiar estructura y dimensiones.
‘pivot_longer()‘	Reemplaza a la función ‘gather()‘ en versiones anteriores del paquete.
‘pivot_wider()‘	Reemplaza a la función ‘spread()‘ en versiones anteriores del paquete.
Nested data	Operación: Anidar datos o extraer datos anidados.
‘nest()‘	Permite anidar tablas tidy data dentro de otras, de modo que una columna contenga una lista de tablas.
‘unnest()‘	Implementa la operación contraria a ‘nest()‘ extrayendo las data frames anidadas.
Rectangling	Operación: Tratamiento de lista de datos anidadas (JSON, XML).
‘unnest_longer()‘	Toma cada elemento de una columna-lista y lo convierte en una nueva fila.
‘unnest_wider()‘	Toma cada argumento de una columna lista y lo convierte en una nueva columna.

6.3. Preparación de datos con tidyverse: tidyr

`'unnest_auto()'` Intenta adivinar la operación apropiada para preparar los datos.

La página web del paquete `tidyr` <https://tidyr.tidyverse.org/> incluye varios artículos y un enlace a la hoja resumen actualizada con la explicación de cada función incluida.

Nosotros vamos a incluir, a modo de ejemplo, dos casos de uso de las herramientas `pivot_longer` y `pivot_wider`. En la página web del paquete `tidyr`, el artículo *Pivoting* describe varios ejemplos utilizando datasets de casos reales. Sin embargo, para entender mejor estas dos funciones vamos a emplear otro ejemplo, extraído del *R Cookbook* de W. Chang¹:

```
# En este ejemplo se usa la función read.table(), que
# permite leer rápidamente datos escritos directamente
# como strings en la consola o en nuestros scripts.

# Ejemplo de tabla en formato wide
experiment_wide <- read.table(header=TRUE, text='
  subject sex control cond1 cond2
    1    M     7.9  12.3  10.7
    2    F     6.3  10.6  11.1
    3    F     9.5  13.1  13.8
    4    M    11.5  13.4  12.9
')

# Ejemplo de tabla en formato long
experiment_long <- read.table(header=TRUE, text='
  subject sex condition measurement
    1    M   control         7.9
    1    M   cond1         12.3
    1    M   cond2         10.7
')
```

¹http://www.cookbook-r.com/Manipulating_data/Converting_data_between_wide_and_long_format/

6. Tidy data

```
      2   F   control      6.3
      2   F    cond1     10.6
      2   F    cond2     11.1
      3   F   control      9.5
      3   F    cond1     13.1
      3   F    cond2     13.8
      4   M   control     11.5
      4   M    cond1     13.4
      4   M    cond2     12.9
    ')
```

Veamos cómo cambiar la primera tabla (*wide*), que además no cumple con las normas de tidy data, al formato de la segunda tabla (*long*), que además sí cumple con las normas de tidy data:

```
# Usamos la función pivot_longer()
# Podemos consultar sus argumentos escribiendo ?pivot_longer en consola R
# Argumentos:
# - cols: columnas de la tabla original que vamos a colapsar en una única co.
# - names_to: nombre de la columna que contendrá como valores los nombres de
# columnas colapsadas, procedentes de la tabla original.
# - values_to: nombre de la columna que contendrá como valores los valores de
# columnas colapsadas, procedentes de la tabla original.
tib_exp_long <-
  experiment_wide %>%
  pivot_longer(cols = c('control', 'cond1', 'cond2'), names_to = 'condition',
               values_to = 'measurement')
tib_exp_long
```

```
# A tibble: 12 x 4
  subject sex   condition measurement
  <int> <chr> <chr>          <dbl>
```


6.3. Preparación de datos con tidyverse: tidyr

1	1 M	control	7.9
2	1 M	cond1	12.3
3	1 M	cond2	10.7
4	2 F	control	6.3
5	2 F	cond1	10.6
6	2 F	cond2	11.1
7	3 F	control	9.5
8	3 F	cond1	13.1
9	3 F	cond2	13.8
10	4 M	control	11.5
11	4 M	cond1	13.4
12	4 M	cond2	12.9

De forma equivalente, se puede pasar de una tabla en formato *long* a una en formato *wide*. ¿Por qué es útil esta transformación? En primer lugar porque, aunque los paquetes de tidyverse esperan tablas de datos que cumplan con las reglas de tidy data, muchos otros paquetes de R no lo esperarán. Antes al contrario, nos obligan a formatear las tablas en una estructura *wide*. Esto es habitual, por ejemplo, en datos para análisis en ciencias sociales de sucesiones de eventos a un mismo individuo (como encuestas de seguimiento realizadas cada cierto intervalo). Otro motivo es que, en ocasiones, no podemos transformar una tabla *messy data* en una tabla *tidy data* en un solo paso. En su lugar, tenemos que hacer varias transformaciones en ambos sentidos hasta completar el proceso.

En nuestro ejemplo, vemos cómo volver a dejar los datos en el formato *wide* original:

```
# Usamos la función pivot_wider()
# Podemos consultar sus argumentos escribiendo ?pivot_wide en consola R
# Argumentos:
# -
tib_exp_wide <-
  experiment_long %>%
```

6. Tidy data

```
pivot_wider(names_from = 'condition', values_from = 'measurement')
tib_exp_wide
```

```
# A tibble: 4 x 5
  subject sex    control cond1 cond2
  <int> <chr>    <dbl> <dbl> <dbl>
1       1 M        7.9   12.3   10.7
2       2 F        6.3   10.6   11.1
3       3 F        9.5   13.1   13.8
4       4 M       11.5   13.4   12.9
```

La siguiente figura ilustra de forma más explícita el proceso de conversión de formato *wide* a formato *long*:

Una vez que hemos garantizado que los datos de nuestro proyecto son tidy data, podemos emplear las potentes herramientas de consulta, manipulación y actualización de datos que nos proporciona el paquete **dplyr**. No obstante, en ocasiones también tendremos que utilizar estas funciones en combinación con las de **tidyr** para resolver problemas complicados de preparación de datos.

6.4. Consulta, manipulación y actualización de datos: dplyr

El paquete **dplyr** proporciona dentro de tidyverse todas las funciones para consulta y manipulación de datos en nuestras tablas. En general, la lista de funciones proporcionada se asemeja (a propósito) a la lista de funciones que podemos encontrar en lenguajes de manipulación de datos en sistemas de bases de datos, tales como el conocido SQL (Structured Query Language). Sin embargo, para facilitar la accesibilidad a usuarios sin muchos conocimientos sobre bases de datos, los nombres de las funciones de **dplyr** suelen ser más claros y explícitos.

6.4.1. Funciones principales en dplyr

La Tabla @ref(tab:dplyr-funcs) resume las principales funciones del paquete dplyr.

Funciones	Descripción
Básicas	Operaciones básicas de recuperación de datos y creación de nuevas variables.
‘select()’	Selecciona las columnas de la tabla que se van a devolver como resultado.
‘filter()’	Filtra filas de la tabla, solo se devuelven las que cumplan las condiciones indicadas.
‘summarise()’	Calcula estadísticos resumen sobre las columnas de la tabla que se van a devolver.
‘arrange()’	Ordena las filas que se van a devolver, según los valores de una o varias variables.
‘mutate()’	Crea nuevas columnas a partir de valores de otras columnas ya presentes en la tabla.
Agrupación	Indica que las operaciones de sucesivos pasos se calcularán por grupos, identificando los grupos.
‘group_by()’	Agrupar por el valor de una o varias variables.
Funciones de dos (o más) tablas	Permiten combinar valores de dos tablas (operaciones con conjuntos, álgebra relacional).
Mutating joins	Combinan dos o más tablas atendiendo a valores en las columnas.
‘inner_join()’	Combina dos o más tablas, dejando solo las filas con valores coincidentes en las columnas indicadas.
‘left_join()’	Combina dos tablas, mostrando todos los valores de una o varias columnas de la primera tabla.
‘right_join()’	Combina dos tablas, mostrando todos los valores de una o varias columnas de la segunda tabla.
‘full_join()’	Realiza el producto cartesiano entre las filas de las dos tablas, es decir, se devuelven todas las combinaciones posibles.
Filtering joins	Unen dos o más tablas pero devolviendo solo filas de la primera tabla.
‘semi_join()’	Devuelve datos de la primera tabla que tengan coincidencias en la segunda tabla.
‘anti_join()’	Devuelve datos de la primera tabla que <i>no</i> tengan coincidencias en la segunda tabla.
Operaciones con conjuntos	Operaciones con conjuntos
‘union()’	Se unen todas las filas de ambas tablas (unión de conjuntos).
‘intersect()’	Solo se devuelven las filas con valores coincidentes en ambas tablas (intersección).
‘setdiff()’	Solo se devuelven las filas con valores <i>no</i> coincidentes en ambas tablas (diferencia).

En la página general de “hojas de referencia” (coloquialmente llamadas

6. Tidy data

chuletas o *cheatsheets* en inglés) de RStudio, podemos encontrar una completa referencia resumen de todas las funciones principales incluidas en `dplyr`, describiendo qué argumentos recibe cada una y ejemplos de uso (enlace descarga)

6.4.2. Ejemplos de uso de dplyr

Seguidamente, presentamos varios ejemplos de utilización de funciones básicas de `dplyr`. Únicamente trataremos aquí funciones básicas de consulta de tablas de datos, así como funciones simples de combinación de valores en dos tablas.

En estos ejemplos utilizamos el dataset `car::UN`, con datos de 213 ubicaciones, entre 2009 y 2011, la mayoría de países miembros de la ONU. Para poder acceder a este dataset en nuestro código, tenemos que cargar primero el paquete `car`. A partir de entonces, tenemos acceso automático a todos los datasets precargados en el paquete, incluido `UN`:

```
library(car)
# Ahora UN está disponible
colnames(UN)
```

```
[1] "region"      "group"      "fertility"   "ppgdp"
[5] "lifeExpF"    "pctUrban"    "infantMortality"
```

```
# Resumen básico de todas las variables
# Prestemos atención a que hay valores faltantes (NA's) en todas
# las columnas
summary(UN)
```

```
      region      group      fertility      ppgdp
Africa   :53  oecd   : 31  Min.    :1.134  Min.    : 114.8
```

6.4. Consulta, manipulación y actualización de datos: dplyr

```

Asia      :50   other :115   1st Qu.:1.754   1st Qu.: 1283.0
Europe    :39   africa: 53   Median :2.262   Median : 4684.5
Latin Amer:20   NA's  : 14   Mean  :2.761   Mean  : 13012.0
Caribbean :17           3rd Qu.:3.545   3rd Qu.: 15520.5
(Other)   :20           Max.   :6.925   Max.   :105095.4
NA's      :14           NA's   :14     NA's   :14

  lifeExpF      pctUrban      infantMortality
Min.   :48.11   Min.     : 11.00   Min.     : 1.916
1st Qu.:65.66   1st Qu.: 39.00   1st Qu.: 7.019
Median :75.89   Median : 59.00   Median : 19.007
Mean   :72.29   Mean   : 57.93   Mean   : 29.440
3rd Qu.:79.58   3rd Qu.: 75.00   3rd Qu.: 44.477
Max.   :87.12   Max.   :100.00   Max.   :124.535
NA's   :14     NA's    :14     NA's    :6

```

Las variables de cada columna en UN son las siguientes:

- **region:** Variable categórica que indica la región de la ubicación que se describe en dicha fila. Posibles valores: **Africa**, **Asia**, **Caribbean**, **Europe**, **Latin Amer**, **North America**, **NorthAtlantic**, **Oceania**.
- **group:** Un factor con 3 posibles niveles:
 - **oecd:** Para países miembros de la OCDE (OECD en inglés).
 - **africa:** Para países en dicho continente; no hay países OECD en este grupo.
 - **other:** Categoría que aglutina a lugares que no pertenecen a ninguno de los dos grupos anteriores.
- **fertility:** Ratio de fertilidad total, medido en número de hijos por mujer.
- **ppgdp:** Productor interior bruto per cápita (PPGDG en inglés), en dólares estadounidenses.

6. Tidy data

- `lifeExpF`: Esperanza de vida de las mujeres, medida en años.
- `pctUrban`: Porcentaje de suelo urbanizado en dicho lugar.
- `infantMortality`: Muertes infantiles antes de cumplir 1 año por cada 1.000 nacimientos con vida.

Veamos ahora algunos ejemplos de utilización de las funciones más habituales en `dplyr()`. Antes de comenzar, creamos un objeto tibble a partir del `data_frame` original.

```
UN_tibble <- as_tibble(UN, rownames = NA) # Mantener nombres de fila
```

6.4.2.1. Selección de columnas con `select()`

- Selección de columnas por nombre.
 - *Recupera únicamente los datos de fertilidad y PPGDP de todos los países del estudio.*

```
# Utilizamos un select para recuperar las columnas por nombre
UN_tibble %>%
  select("fertility", "ppgdp") %>%
  head()
```

```
# A tibble: 6 x 2
  fertility ppgdp
    <dbl>   <dbl>
1     5.97    499
2     1.52  3677.
3     2.14  4473
4      NA     NA
5     5.14  4322.
6      2   13750.
```

6.4. Consulta, manipulación y actualización de datos: dplyr

- Selección de columnas excluyendo algunas.
 - Recupera todas las columnas excepto el porcentaje de suelo urbanizado.
 - Repetimos pero ahora eliminando, además, la esperanza de vida femenina.

```
# Quitamos suelo urbanizado
UN_tibble %>%
  select(-"pctUrban") %>%
  head()
```

```
# A tibble: 6 x 6
  region    group fertility ppgdp lifeExpF infantMortality
<fct>    <fct>    <dbl>   <dbl>   <dbl>         <dbl>
1 Asia      other     5.97    499     49.5          125.
2 Europe    other     1.52   3677.    80.4           16.6
3 Africa    africa     2.14   4473     75             21.5
4 <NA>      <NA>      NA       NA      NA             11.3
5 Africa    africa     5.14   4322.    53.2           96.2
6 Caribbean other      2    13750.    81.1            NA
```

```
# Ahora, quitamos además esperanza de vida
UN_tibble %>%
  select(-c("pctUrban", "lifeExpF")) %>%
  head()
```

```
# A tibble: 6 x 5
  region    group fertility ppgdp infantMortality
<fct>    <fct>    <dbl>   <dbl>         <dbl>
1 Asia      other     5.97    499          125.
2 Europe    other     1.52   3677.          16.6
3 Africa    africa     2.14   4473          21.5
```

6. Tidy data

4	<NA>	<NA>	NA	NA	11.3
5	Africa	africa	5.14	4322.	96.2
6	Caribbean	other	2	13750.	NA

- Selección de columnas por posición.
 - *Selecciona la primera, tercera y cuarta columna de la tabla.*

```
# Podemos utilizar argumentos numéricos para indicar posición de columna
# En ese caso, no llevan comillas.
UN_tibble %>%
  select(1,3,4) %>%
  head()
```

```
# A tibble: 6 x 3
  region    fertility ppgdp
  <fct>      <dbl>   <dbl>
1 Asia        5.97    499
2 Europe      1.52   3677.
3 Africa      2.14   4473
4 <NA>        NA      NA
5 Africa      5.14   4322.
6 Caribbean   2     13750.
```

- Selección de un rango consecutivo de columnas.
 - *Selecciona las columnas entre grupo de país y esperanza de vida femenina.*
 - *Selecciona las tres primeras columnas de la tabla.*

```
# Usamos la notación de rango consecutivo con nombres de columnas
# Atención: ambos extremos del rango están incluidos en la selección
UN_tibble %>%
  select("group":"lifeExpF") %>%
  head()
```


6.4. Consulta, manipulación y actualización de datos: dplyr

```
# A tibble: 6 x 4
  group fertility ppgdp lifeExpF
  <fct>      <dbl> <dbl>   <dbl>
1 other      5.97   499     49.5
2 other      1.52  3677.    80.4
3 africa     2.14  4473     75
4 <NA>       NA      NA      NA
5 africa     5.14  4322.    53.2
6 other      2     13750.   81.1
```

```
# Para la segunda usamos la misma notación, pero con números de columnas
UN_tibble %>%
  select(1:3) %>%
  head()
```

```
# A tibble: 6 x 3
  region    group fertility
  <fct>    <fct>      <dbl>
1 Asia    other      5.97
2 Europe  other      1.52
3 Africa  africa     2.14
4 <NA>    <NA>       NA
5 Africa  africa     5.14
6 Caribbean other      2
```

- Selección de columnas programáticamente (guardando nombres en una variable).
 - *Simula que tienes un vector de nombres de columnas, que por ejemplo puedas haber calculado en un código anterior. Usa los valores de ese vector para seleccionar las columnas correspondientes de la tabla.*

6. Tidy data

```
# Imaginemos que el contenido del vector después de las operaciones es este
nombres_cols = c("fertility", "ppgdp", "infantMortality")

UN_tibble %>%
  select(nombres_cols) %>%
  head()
```

Warning: Using an external vector in selections was deprecated in tidyselect. Please use `all_of()` or `any_of()` instead.

Was:

```
data %>% select(nombres_cols)
```

Now:

```
data %>% select(all_of(nombres_cols))
```

See <https://tidyselect.r-lib.org/reference/faq-external-vector.html>.

```
# A tibble: 6 x 3
  fertility ppgdp infantMortality
    <dbl>   <dbl>         <dbl>
1     5.97   499          125.
2     1.52 3677.          16.6
3     2.14 4473          21.5
4     NA    NA           11.3
5     5.14 4322.          96.2
6      2 13750.           NA
```

6.4.2.2. Selección de filas con `filter()`

- Filtrado de filas usando expresiones lógicas (booleanas).
 - *Selecciona las ubicaciones que pertenezcan al grupo `other` y tengan un ratio de fertilidad mayor de 2.1.*

6.4. Consulta, manipulación y actualización de datos: dplyr

- *Selecciona las ubicaciones que pertenezcan a la región de Europa o Latinoamérica con un producto interior bruto per cápita mayor de 25.000 dólares.*

```
# Usamos los operadores de comparación lógica de R para imponer las
# condiciones de selección de datos
UN_tibble %>%
  filter(group == "other" & fertility > 2.1) %>%
  head()
```

```
# A tibble: 6 x 7
  region    group fertility  ppgdp lifeExpF pctUrban infantMortality
  <fct>    <fct>    <dbl>   <dbl>   <dbl>   <dbl>         <dbl>
1 Asia      other     5.97   499     49.5     23           125.
2 Latin Amer other     2.17  9162.    79.9     93           12.3
3 Asia      other     2.15  5638.    73.7     52           37.6
4 Asia      other     2.43 18184.    76.1     89            6.66
5 Asia      other     2.16   670.    70.2     29           41.8
6 Latin Amer other     2.68  4496.    77.8     53           16.2
```

```
UN_tibble %>%
  filter((region %in% c("Europe", "Latin Amer")) & ppgdp > 25000) %>%
  head()
```

```
# A tibble: 6 x 7
  region group fertility  ppgdp lifeExpF pctUrban infantMortality
  <fct>  <fct>    <dbl>   <dbl>   <dbl>   <dbl>         <dbl>
1 Europe oecd     1.35 45159.    83.6     68           3.71
2 Europe oecd     1.84 43815.    82.8     97           3.74
3 Europe oecd     1.88 55830.    81.4     87           3.91
4 Europe oecd     1.88 44502.    83.3     85           2.78
5 Europe oecd     1.99 39546.    84.9     86           3.34
6 Europe oecd     1.46 39857.    83.0     74           3.49
```

6. Tidy data

6.4.2.3. Resúmenes de datos con `summarise()`

- Estadísticos resumen para todos los valores de la tabla, o para los grupos que hayan sido creados si antes hemos llamado a `group_by()`.
 - *Calcula el promedio de fertilidad y de PPGDP de todos los países que pertenecen a la OCDE.*
 - *Calcula el número de países que hay en cada región de la muestra* (**Nota:** consulta la sección “Summary Functions” en la segunda página de la hoja resumen de `dplyr` de RStudio para aprender más funciones útiles que puedes usar con `summarise()`).

```
# ¿Qué ocurre si no ponemos el argumento na.rm = TRUE? ¿Por qué?
UN_tibble %>%
  summarise(avg_fertility = mean(fertility, na.rm = TRUE),
            avg_ppgdp = mean(ppgdp, na.rm = TRUE))
```

```
# A tibble: 1 x 2
  avg_fertility avg_ppgdp
      <dbl>      <dbl>
1         2.76    13012.
```

```
UN_tibble %>%
  filter(!is.na(region)) %>% # Quitamos NA's
  group_by(region) %>% # Pedimos el resumen agrupados por región
  summarise(num_countries = n())
```

```
# A tibble: 8 x 2
  region      num_countries
  <fct>          <int>
1 Africa             53
2 Asia              50
3 Caribbean          17
```

6.4. Consulta, manipulación y actualización de datos: dplyr

4 Europe	39
5 Latin Amer	20
6 North America	2
7 NorthAtlantic	1
8 Oceania	17

6.4.2.4. Agrupamiento de datos con `group_by()`

- Agrupando los resultados respecto a valores de una columna.
 - ¿Cuál es la media del producto interior bruto per cápita del conjunto de países dentro de cada grupo?

```
UN_tibble %>%  
  filter(!is.na(group)) %>% # Quitamos NA's en grupo  
  group_by(group) %>% # Pedimos el resumen agrupados por región  
  summarise(avg_ppgdp_group = mean(ppgdp, na.rm = TRUE)) # Filtramos NA's en ppgdp
```

```
# A tibble: 3 x 2  
  group avg_ppgdp_group  
  <fct>         <dbl>  
1 oecd          37761.  
2 other          11181.  
3 africa         2509.
```

- Agrupando los resultados respecto a valores de varias columnas.
 - *¿Cuál es el promedio de fertilidad en los países de cada grupo y, dentro del mismo, según la región a la que pertenecen?

```
UN_tibble %>%  
  filter(!is.na(group) & !is.na(region)) %>% # Quitamos NA's en grupo  
  group_by(group, region) %>% # Pedimos el resumen agrupados por grupo, región  
  summarise(avg_fertility = mean(fertility, na.rm = TRUE)) # Filtramos NA's en ppgdp
```

6. Tidy data

``summarise()`` has grouped output by 'group'. You can override using the ``groups`` argument.

```
# A tibble: 12 x 3
# Groups:   group [3]
  group region      avg_fertility
  <fct> <fct>          <dbl>
1 oecd  Asia           2.12
2 oecd  Europe          1.66
3 oecd  Latin Amer      2.03
4 oecd  North America   1.88
5 oecd  Oceania         2.04
6 other Asia        2.45
7 other Caribbean  2.01
8 other Europe      1.50
9 other Latin Amer   2.47
10 other NorthAtlantic 2.22
11 other Oceania     3.24
12 africa Africa      4.24
```

6.4.2.5. Ordenación de datos con `arrange()`

- Ordenación de resultados por valores de una columna. Se hace por defecto de menor a mayor. Si queremos ordenar a la inversa usamos `desc()`.
 - *Obtén los 5 países de Europa que tienen mayor superficie urbanizada.*

```
# Después de la consulta, usamos head() para restringir el número de filas
# En el siguiente capítulo presentamos top_n() para el mismo uso
UN_tibble %>%
  select("region", "ppgdp") %>%
```

6.4. Consulta, manipulación y actualización de datos: dplyr

```
filter(region == "Europe") %>%  
arrange(desc(ppgdp)) %>%  
head(5)
```

```
# A tibble: 5 x 2  
  region    ppgdp  
  <fct>    <dbl>  
1 Europe 105095.  
2 Europe  84589.  
3 Europe  68880.  
4 Europe  55830.  
5 Europe  48906.
```

- Ordenación de resultados por valores de varias columnas.
 - *Ordena los resultados de promedio de fertilidad de mujeres por grupo y región, de forma que primero aparezcan las regiones en orden alfabético y, para cada región, los grupos en orden alfabético.*

```
UN_tibble %>%  
  select(region, group, fertility) %>%  
  filter(!is.na(group) & !is.na(region)) %>%  
  group_by(group, region) %>%  
  summarise(avg_fertility = mean(fertility, na.rm = TRUE)) %>%  
  select(2,1,3) %>% # Volvemos a llamar a select para cambiar el orden de columnas  
  arrange(region, group) # Aquí ordenamos por región y dentro de cada una por grupo
```

`summarise()` has grouped output by 'group'. You can override using the
`.groups` argument.

```
# A tibble: 12 x 3
```

6. Tidy data

```
# Groups:   group [3]
  region      group avg_fertility
  <fct>       <fct>      <dbl>
1 Africa      africa      4.24
2 Asia        oecd        2.12
3 Asia        other       2.45
4 Caribbean   other       2.01
5 Europe      oecd        1.66
6 Europe      other       1.50
7 Latin Amer  oecd        2.03
8 Latin Amer  other       2.47
9 North Amer  oecd        1.88
10 NorthAtlantic other    2.22
11 Oceania    oecd        2.04
12 Oceania    other       3.24
```

6.4.2.6. Nuevas columnas con mutate()

Cuidado: si no almacenamos el resultado de la operación en una nueva variable, la nueva columna creada solo existirá durante la ejecución de la consulta que hemos escrito. Una vez devuelto el resultado, la nueva columna desaparece. **Nunca se altera la tabla de datos original.**

- Creación de una columna nueva a partir de operaciones sobre otra.
 - *Transformar todos los valores de la columna `ppgdp`, calculando su logaritmo natural..*

```
# Transformar el valor de `ppgdp` con log()
UN_tibble %>%
  mutate(log_ppgdp = log(ppgdp)) %>%
  head()
```


6.4. Consulta, manipulación y actualización de datos: dplyr

```
# A tibble: 6 x 8
  region    group fertility ppgdp lifeExpF pctUrban infantMortality log_ppgdp
  <fct>    <fct>    <dbl>  <dbl>  <dbl>   <dbl>      <dbl>      <dbl>
1 Asia     other      5.97   499    49.5    23        125.        6.21
2 Europe   other      1.52  3677.   80.4    53         16.6        8.21
3 Africa   africa     2.14  4473    75      67         21.5        8.41
4 <NA>     <NA>      NA     NA     NA      NA         11.3        NA
5 Africa   africa     5.14  4322.   53.2    59         96.2        8.37
6 Caribbean other      2    13750.   81.1   100         NA         9.53
```

- Creación de una columna nueva combinando valores de varias columnas.
 - *Calcular una nueva columna como el cociente entre mortalidad infantil y fertilidad de las mujeres para cada ubicación.*

```
UN_tibble %>%
  mutate(mort_by_fert = infantMortality/fertility) %>%
  head()
```

```
# A tibble: 6 x 8
  region    group fertility ppgdp lifeExpF pctUrban infantMortality mort_by_fert
  <fct>    <fct>    <dbl>  <dbl>  <dbl>   <dbl>      <dbl>      <dbl>
1 Asia     other      5.97   499    49.5    23        125.        20.9
2 Europe   other      1.52  3677.   80.4    53         16.6        10.9
3 Africa   afri~     2.14  4473    75      67         21.5        10.0
4 <NA>     <NA>      NA     NA     NA      NA         11.3        NA
5 Africa   afri~     5.14  4322.   53.2    59         96.2        18.7
6 Caribbe~ other      2    13750.   81.1   100         NA         NA
```

6.4.2.7. Combinación de tablas con inner_join()

Para esta sección, vamos a crear una segunda tabla de datos ficticios, incluyendo una variable descriptiva más para cada región. Para ello, vamos

6. Tidy data

a usar la función `tribble()` (c.f. <https://tibble.tidyverse.org/>).

```
# Estimaciones: https://www.worldometers.info/world-population/population-by-
UN2 <- tribble (
  ~region, ~population, ~area,
  "Africa",    1308064195, 29648481,
  "Asia",     4601371198, 31033131,
  "Caribbean", 48000000, 1139000,
  "Europe",   747182751, 22134900,
  "Latin Amer", 600120957, 19000378,
  "North America", 360600964, 17651660,
  "NorthAtlantic", 6000000, 1000000,
  "Oceania",  42128035, 8486460
)
UN2$region = factor(UN2$region) # Convertir region a factor
```

- Combinación de dos tablas según los valores comunes de una columna en cada tabla.
 - *Calcula el área total (en km^2) urbanizada en cada región..*

```
# Hay que conectar las filas de las tablas UN y UN2 según el valor de su reg:
# Una vez tenemos la fila con todos los datos, calculamos el producto del
# porcentaje de suelo urbanizado y la superficie total.
# Como los nombres de fila del data.frame original son en principio ignorados
# por un objeto tibble, existen funciones que permiten recuperarlos:
# https://tibble.tidyverse.org/reference/rownames.html.
# Aquí usamos `rownames_to_column()`

UN_tibble %>%
  rownames_to_column(var="location") %>%
  select("location", "region", "pctUrban") %>%
  inner_join(UN2, by="region") %>%
  mutate(urban_km2 = pctUrban/100 * area)
```

6.4. Consulta, manipulación y actualización de datos: dplyr

```
# A tibble: 199 x 6
  location      region    pctUrban population    area urban_km2
  <chr>         <fct>         <dbl>         <dbl>         <dbl>         <dbl>
1 Afghanistan Asia           23 4601371198 31033131 7137620.
2 Albania      Europe           53 747182751 22134900 11731497
3 Algeria      Africa           67 1308064195 29648481 19864482.
4 Angola       Africa           59 1308064195 29648481 17492604.
5 Anguilla     Caribbean       100 48000000 1139000 1139000
6 Argentina    Latin Amer       93 600120957 19000378 17670352.
7 Armenia      Asia            64 4601371198 31033131 19861204.
8 Aruba        Caribbean       47 48000000 1139000 535330
9 Australia    Oceania         89 42128035 8486460 7552949.
10 Austria     Europe          68 747182751 22134900 15051732
# i 189 more rows
```

```
#rmarkdown::paged_table(options = list(rows.print = 15))
```


7. Importación y exportación de datos

7.1. Importación de datos

Hasta ahora, nos hemos basado generalmente en datos que ya están previamente cargados y disponibles en paquetes de R para nuestros ejemplos. Sin embargo, lo habitual en un proyecto es que tengamos que leer los datos de algún tipo de fuente externa a R:

- **Archivo textual:** Los datos pueden estar almacenados en formato libre, pero lo normal es que sigan una pauta para organizar su presentación, según algún **formato estándar** de representación de datos.
 - **CSV** (Comma Separated Values): Los datos se escriben insertando una línea por cada fila de la tabla, y separando los valores de cada columna mediante comas. Los strings suelen ir delimitados por dobles comillas (“). Desgraciadamente, no existe una convención única sobre qué caracteres usar para delimitar strings o separar los valores de cada columna. A veces también se pueden usar puntos y coma, tabuladores, etc. Debemos informarnos sobre el formato empleado o descubrirlo nosotros, inspeccionando algunas líneas del archivo en crudo.
 - **JSON** (JavaScript Object Notation): Es un formato que se ha popularizado bastante, porque lo suelen emplear muchos servicios y APIs web que exportan datos. Es bastante compacto y

7. Importación y exportación de datos

permite representar arrays y listas arbitrarias de objetos. También se puede usar este formato para guardar datos en archivos de texto.

- **Base de datos** (normalmente relacional): Guarda los datos en tablas similares a un `data.frame` (o `tibble`). R proporciona paquetes que permiten conectarnos a muchos tipos de bases de datos para realizar consultas y recuperar información. Normalmente, el resultado que la base de datos envía como respuesta a nuestra consulta se formatea automáticamente como un `data.frame` (lo hacen las propias bibliotecas conectoras de R).
- **Formatos binarios** (Excel, OpenOffice, LibreOffice, etc.): En ocasiones los datos se han creado previamente en un programa ofimático o de otro tipo, y tenemos que importarlos en R. El caso habitual es Excel que, hoy en día, continúa siendo una herramienta muy habitual en gestión y cálculo de datos.

No es posible cubrir toda esta casuística en el tiempo de curso. Por ello, nos vamos a centrar en la lectura de datos CSV y la lectura de datos de Excel para cubrir dos ejemplos muy comunes de importación de datos en R.

7.1.1. Importar datos de archivos de texto: CSV

Para la mayoría de formatos de almacenamiento en archivos de texto, las funciones del paquete `readr`, que también forma parte del núcleo básico de `tidyverse`, son una buena alternativa para cargar de manera fiable nuestros datos en R.

Podemos ejecutar en consola el siguiente comando, para mostrar una lista de todas las funciones incluidas en este paquete:

7.1. Importación de datos

```
help(package="readr")
```

Todas las funciones que comienzan por `readr_` están pensadas para automatizar la lectura de datos de archivos textuales. Por desgracia, no existe un formato universal de representación de datos en CSV. Así que hay diferentes funciones para cada caso. Presentamos en la tabla @ref(tab:funcs-read) las principales.

Tabla 7.1.: (#tab:funcs-read) Funciones típicas de lectura de datos en formato CSV y similares.

Nombre función	Descripción
<code>read_delim()</code>	Función genérica, hay que especificar delimitador.
<code>read_csv()</code>	Asume delimitador es “,” y que “.” separa decimales.
<code>read_csv2()</code>	Asume delimitador es “;” y que “,” separa decimales.
<code>read_tsv()</code>	Asume delimitador es “\t” (tabulador)

Veamos un ejemplo que muestre la potencia de estas funciones. En el subdirectorio `data` del directorio de este manual, tenemos el archivo `ACMETelephoneABT.csv`. Este archivo proviene de un ejemplo de un conocido (y muy recomendable) libro de introducción a la ciencia de datos y el aprendizaje máquina (Kelleher et al., 2015).

Ahora comprobamos lo fácil que es leer los datos, en una sola línea de código R:

```
ACMETelephoneABT <- read_csv("data/ACMETelephoneABT.csv", na = c("", " "))
```

```
Rows: 10000 Columns: 33
```

```
-- Column specification -----
```

```
Delimiter: ","
```

7. Importación y exportación de datos

```
chr  (5): occupation, regionType, marriageStatus, creditRating, creditCard
dbl (24): customer, age, income, numHandsets, handsetAge, currentHandsetPrice,
lg1  (4): children, smartPhone, homeOwner, churn
```

```
i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
class(ACMETelescopeABT) # Comprobamos clase del objeto
```

```
[1] "spec_tbl_df" "tbl_df"      "tbl"        "data.frame"
```

```
dim(ACMETelescopeABT) # Dimensiones de la tabla
```

```
[1] 10000    33
```

```
head(ACMETelescopeABT, n = 75) # 5 páginas de tabla de resultados
```

```
# A tibble: 75 x 33
```

	customer	age	occupation	regionType	marriageStatus	children	income
	<dbl>	<dbl>	<chr>	<chr>	<chr>	<lg1>	<dbl>
1	1000004	26	crafts	town	yes	TRUE	6
2	1000012	36	<NA>	<NA>	yes	TRUE	9
3	1000034	74	professional	town	yes	FALSE	7
4	1000063	30	<NA>	suburban	no	FALSE	6
5	1000085	32	<NA>	town	yes	FALSE	7
6	1000100	40	homemaker	rural	yes	TRUE	7
7	1000121	52	<NA>	<NA>	no	FALSE	3
8	1000135	32	clerical	<NA>	yes	FALSE	5
9	1000143	40	<NA>	<NA>	unknown	FALSE	5
10	1000164	42	<NA>	<NA>	yes	TRUE	4

```
# i 65 more rows
```


7.1. Importación de datos

```
# i 26 more variables: numHandsets <dbl>, handsetAge <dbl>, smartPhone <lgl>,  
#   currentHandsetPrice <dbl>, creditRating <chr>, homeOwner <lgl>,  
#   creditCard <chr>, avgBill <dbl>, avgMins <dbl>, avgrecurringCharge <dbl>,  
#   avgOverBundleMins <dbl>, avgRoamCalls <dbl>, callMinutesChangePct <dbl>,  
#   billAmountChangePct <dbl>, avgReceivedMins <dbl>, avgOutCalls <dbl>,  
#   avgInCalls <dbl>, peakOffPeakRatio <dbl>, ...
```

Conviene resaltar algunos puntos importantes:

- La función `read_csv()` utilizada sin argumentos (como en este ejemplo) intenta usar heurísticos sencillos para *adivinar* el tipo de datos adecuado para formatear los valores de cada columna. Si no ocultamos los avisos, la función nos indica (como vemos arriba en este documento) qué formatos ha utilizado para leer cada una de las columnas. Conviene no ocultar esta información, ya que muchos errores en preparación y limpieza de datos provienen de confusiones con el tipo de datos asignado a una columna.
- Los valores de tipo string (en tibble marcados como `<chr>`) **no se convierten a factores de forma automática**, sino que se dejan como cadena de caracteres. Por tanto, si deseamos que se considere a los valores de alguna columna explícitamente como factor, habrá que especificarlo en el momento de leer los datos, o bien convertirlo explícitamente más adelante. Será más conveniente elegir una u otra opción dependiendo de si tenemos que limpiar intensamente los datos de esa columna o ya vienen en buen estado.
- Es importante especificar qué convención se usa para indicar los datos faltantes en el archivo. Suele ser arbitrario (y, por desgracia, a veces caótico), obligándonos a limpiar los datos siguiendo el método de prueba y error. Por defecto, la función considera el argumento `na = c("", "NA")`, es decir un string vacío o el string “NA” marcan un dato faltante en la tabla. En nuestro caso, bien porque nos han avisado o por una inspección preliminar del contenido del archivo

7. Importación y exportación de datos

sabemos que, en su lugar, los datos faltantes se marcan con string vacío o con espacio en blanco (" ").

Con esto hemos leído de forma aparentemente sencilla los datos de un archivo CSV. En este caso, hemos usado `read_csv()` porque los autores son británicos, y por tanto usan la convención de separar en cada fila los valores de cada columna mediante “,” y usar “.” como caracter para delimitar la parte decimal de un número.

Sin embargo, más adelante volveremos a tratar este dataset y comprobaremos que, como suele ser habitual en un caso real, la lectura inicial de datos no es más que el primer paso de un proceso de **limpieza de datos**, que suele **consumir en torno a un 80% del tiempo** de trabajo en el proyecto para el científico de datos.

Por último, es importante resaltar que en RStudio podemos también usar de forma equivalente una **interfaz gráfica** para diseñar interactivamente el comando de lectura del archivo CSV que vamos a ejecutar. Para ello, en el panel multipropósito (normalmente arriba a la derecha), pinchamos en la pestaña “*Environment*” y pulsamos el botón “*Import Dataset*”. Después, seleccionamos la opción “*From Text (readr)...*” y aparece un cuadro de diálogo. En la parte superior, pinchando en el botón “*Browse*” podemos navegar el disco para seleccionar el archivo a cargar.

Al seleccionarlo, en el cuadro de diálogo aparece una previsualización del que será el resultado de la lectura una vez ejecutado el comando, que se va construyendo en la parte inferior derecha del cuadro diálogo. Este comando se ejecutará automáticamente en consola al confirmar el proceso, pulsando el botón “*Import*”. En la parte superior de las columnas, podemos pinchar en los títulos de columna para desplegar una larga lista con todos los tipos de datos disponibles para formatear dicha columna. Si elegimos un nuevo tipo de datos que no es el considerado por defecto por la función, el comando de lectura se actualiza automáticamente para formatear la columna según el nuevo tipo de dato elegido.

7.1.2. Importar datos de archivos Excel

Importar datos desde un archivo Excel, siempre que no contenga características especiales (macros, multicolumnas o multifilas, automatismos, etc.), es también fácilmente abordable mediante el paquete `readxl` y la función `read_excel()` incluida en el mismo.

En este caso también se puede teclear el comando en consola (o en un script de R), o bien usar el cuadro diálogo interactivo para diseñar el comando de lectura del fichero de Excel. A esta segunda opción se accede también pulsando en la pestaña “*Environment*” del panel multipropósito el botón “*Import Dataset*”, y seleccionando “*From Excel...*”.

Enviamos a los lectores interesados en probar un ejemplo de lectura de ficheros Excel a realizar el ejercicio guiado propuesto en el capítulo 4 del libro en línea “Modern Dive into R” (sec. 4.1.2).

7.1.3. Otras herramientas de importación de datos

En este curso solo cubrimos los casos más típicos y sencillos de importación de datos. Sin embargo, este paso puede acarrear muchos quebraderos de cabeza a los científicos de datos que deben abordarlo. Especialmente difícil resulta la lectura rápida y eficiente de archivos muy largos, que pueden contener cientos de miles o decenas de millones de líneas (o incluso más), cada una representando una fila de la futura tabla con todos los valores para cada columna.

Por este motivo, presentamos aquí un par de alternativas para que el lector que se enfrente en alguna ocasión a este reto tenga pistas sobre dónde empezar a documentarse para resolver el problema:

- El paquete `vroom` (cuidado, no confundir con `broom`), es una alternativa reciente para lectura/escritura de datos extremadamente rápida

7. Importación y exportación de datos

y eficiente en R. Referimos al lector interesado a la viñeta de introducción a `vroom` para más detalles. Entre las funciones más resaltables tenemos:

- Lectura automatizada de múltiples archivos.
 - Lectura de archivos comprimidos.
 - Lectura de archivos en ubicaciones remotas (con una URL).
- Otro proyecto que merece la pena destacar es Apache Arrow, que aspira a convertirse en el formato universal, independiente de plataforma para representación y serialización de datos en memoria. Cuenta con paquetes y librerías de enlace en muchos lenguajes de programación, incluido el paquete `arrow` en R. Su gran ventaja es que se emplea un formato universal de representación de datos extremadamente eficiente, que se mantiene inalterado con independencia del lenguaje de programación o plataforma de ejecución que empleemos.

Parte III.

Ejemplo práctico

8. Análisis exploratorio de datos

El Análisis Exploratorio de Datos (*Exploratory Data Analysis* o EDA) (Tukey, 1977) es un conjunto de técnicas estadísticas y de visualización de datos que permiten identificar las características más relevantes de un conjunto de datos sin intentar aplicar un modelo o algoritmo en particular.

A continuación, veremos varios ejemplos que ilustran cómo realizar operaciones básicas de EDA con R.

8.1. Datos de ejemplo

Utilizaremos ejemplos del paquete `agriTutorial` (Rodney Edmondson et al., 2020), que incluye conjuntos de datos y ejemplos de modelos sobre experimentos en agricultura. También usaremos el paquete R `agridat` (Wright, 2024), que contiene conjuntos de datos provenientes de publicaciones relacionadas con la agricultura, incluyendo cultivos de campo, cultivos arbóreos o estudios con animales, entre otros.

Paquete `agriTutorial`

- `greenrice`: Datos de un experimento sobre absorción de nitrógeno (N), medido en g/maceta en un experimento de doble factor en invernadero (Gomez & Gomez, 1984). Se considera la duración del estrés hídrico (W) y el nivel de aplicación de nitrógeno. El experimento se llevó a cabo con cuatro niveles de estrés hídrico (0, 10, 20 y 40 días)

8. Análisis exploratorio de datos

como tratamientos de parcela principal (*main-plot treatment*) y cuatro dosis de nitrógeno (0, 90, 180 y 270 kg/ha) como tratamientos de subparcela (*sub-plot treatment*). Las parcelas principales se aleatorizaron en cuatro bloques completos (cada bloque incluye todos las posibles combinaciones de tratamientos).

- **rice:** Datos de un experimento con tres prácticas de gestión (**minimum**, **optimum**, **intensive**), cinco cantidades distintas de fertilizante de nitrógeno (N) (0, 50, 80, 110, 140 kg/ha) y tres variedades de planta (**V1**, **V2**, **V3**) (Gomez & Gomez, 1984). El experimento sigue un diseño de parcelas divididas (*split-plot design*), donde la variedad de planta y el tipo de gestión son factores de tratamiento cualitativos y la cantidad de fertilizante de nitrógeno es un factor de tratamiento cuantitativo.

Paquete agridat

- **nass.barley**, **nass.corn**, **nass.rice**, **nass.soybean** y **nass.wheat:** Cinco conjunto de datos con valores de producción y superficie (en acres) cosechados en cada estado de EE.UU. para los cultivos agrícolas más importantes en este país, entre aproximadamente el año 1900 y el 2011. [*Fuente:* United States Department of Agriculture, National Agricultural Statistics Service. <https://quickstats.nass.usda.gov/>]

8.2. Carga de datos y esquema (tipos de datos)

Los conjuntos de datos de ejemplo ya se encuentran disponibles en cada uno de los paquetes mencionados. Sin embargo, vamos a hacer una prueba para verificar cómo podemos escribir y leer datos en R, por ejemplo utilizando un fichero CSV (muy común para almacenamiento de datos estructurados).

8.2. Carga de datos y esquema (tipos de datos)

Tip

Podemos utilizar la función `write.csv()` para escribir datos en un fichero en formato CSV que almacenemos en una ubicación de nuestro disco.

```
library(agridat)
library(agriTutorial)
# Carga de datos en memoria
data("nass.corn")
data("nass.corn")
data("greenrice")
# Escritura de datos a un fichero CSV
write.csv(nass.barley, "data/nass_barley.csv", row.names = F)
write.csv(nass.corn, "data/nass_corn.csv", row.names = F)
write.csv(greenrice, "data/greenrice.csv", row.names = F)
```

Tip

Podemos utilizar la función `read_csv()` del paquete `readr` para leer datos de un fichero en formato CSV almacenado en una ubicación de nuestro disco.

La lectura de datos será más rápida si indicamos explícitamente el tipo de dato que almacena cada columna mediante la función `cols()` y utilizando las funciones que denotan cada tipo de datos específico: `col_integer()`, `col_character()`, `col_double()`, etc.).

```
library(readr)
library(tibble)
# Leemos los datos de los ficheros CSV creados antes
nass_barley <- read_csv("data/nass_barley.csv", col_types = cols(col_integer(), col_character(),
                                                                col_double(), col_double()))
```

8. Análisis exploratorio de datos

```
nass_corn <- read_csv("data/nass_corn.csv", col_types = cols(col_integer(), col_double(), col_double()))
green_rice <- read_csv("data/greenrice.csv", col_types = cols(col_double(), col_double(), col_double(), col_integer(), col_integer(), col_integer()))
```

Una vez que hemos cargado los datos de un paquete R o los hemos leído de un fichero, podemos imprimir en pantalla información acerca de la tabla de datos para comprobar, por ejemplo, que todas las columnas se han identificado correctamente y el tipo de dato de cada columna es correcto.

Tip

Los datos en muchos paquetes R están en formato `data.frame`, menos potente que los objetos `tibble` creados mediante funciones como `read_csv()`. Podemos crear automáticamente un objeto `tibble` a partir de un `data.frame` con la función `tibble()`:

```
my_tibble <- tibble(greenrice)
```

Tip

Para comprobar los metadatos de una tabla en formato `tibble` podemos usar la función `str()`.

```
str(nass_barley)
```

```
spc_tbl_ [4,839 x 4] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ year : int [1:4839] 1866 1866 1866 1866 1866 1866 1866 1866 1866 1866 ...
 $ state: chr [1:4839] "Connecticut" "Illinois" "Indiana" "Iowa" ...
 $ acres: num [1:4839] 1000 96000 11000 66000 2000 10000 34000 7000 21000 20000 ...
 $ yield: num [1:4839] 22.5 23.4 23 22 23 23.5 21.5 25.5 26 26 ...
- attr(*, "spec")=
 .. cols(
```

8.3. Operaciones básicas

```
.. year = col_integer(),  
.. state = col_character(),  
.. acres = col_double(),  
.. yield = col_double()  
.. )  
- attr(*, "problems")=<externalptr>
```

8.3. Operaciones básicas

```
library(dplyr) # Operaciones de selección y filtrado de datos  
library(tidyr) # Operaciones de transformación y procesado de datos
```

8.3.1. Operaciones de selección

Selección de columnas

Para seleccionar determinadas columnas (y todas sus filas) usamos la función `select()`.

```
green_rice_uptake <- select(green_rice, Replicate, Main, uptake)  
str(green_rice_uptake)
```

```
tibble [64 x 3] (S3: tbl_df/tbl/data.frame)  
$ Replicate: int [1:64] 1 1 1 1 1 1 1 1 1 1 ...  
$ Main      : int [1:64] 1 1 1 1 2 2 2 2 3 3 ...  
$ uptake    : num [1:64] 1.033 0.506 0.692 0.254 0.154 ...
```

Filtrado de filas

8. Análisis exploratorio de datos

Podemos añadir condiciones adicionales de filtrado (mediante expresiones booleanas) que se utilicen para recuperar sólo las filas de esa columna o columnas que cumplan cierta condición.

```
corn_MI <- filter(nass_corn, state == "Michigan")
head(corn_MI)
```

```
# A tibble: 6 x 4
  year state    acres yield
  <int> <chr>    <dbl> <dbl>
1  1866 Michigan 480000  32
2  1867 Michigan 500000  36
3  1868 Michigan 560000  33
4  1869 Michigan 540000  28
5  1870 Michigan 570000  39
6  1871 Michigan 600000  33.5
```

También se pueden imponer simultáneamente varias condiciones de filtrado. En ese caso, es común utilizar el operador `%in%`, para comprobar valores dentro de un conjunto finito de opciones válidas.

```
corn_pnw <- filter(nass_corn, state %in% c("Idaho", "Oregon", "Washington"))
head(corn_pnw)
```

```
# A tibble: 6 x 4
  year state    acres yield
  <int> <chr>    <dbl> <dbl>
1  1869 Oregon  3000  25
2  1870 Oregon  3000  23
3  1871 Oregon  4000  25
4  1872 Oregon  4000  26.5
5  1873 Oregon  4000  26
6  1874 Oregon  4000  25
```

8.3. Operaciones básicas

Por último, también se pueden usar valores obtenidos como resultado de otras operaciones. Por ejemplo, veamos cómo recuperar sólo las filas que tengan un valor de superficie menor que la mediana de esa variable.

```
corn_low_surf <- filter(nass_corn, acres < median(nass_corn$acres))
head(corn_low_surf)
```

```
# A tibble: 6 x 4
  year state      acres yield
  <int> <chr>    <dbl> <dbl>
1  1866 Arkansas 280000    18
2  1866 California 42000    28
3  1866 Connecticut 57000    34
4  1866 Delaware 200000    23
5  1866 Florida 125000     9
6  1866 Kansas 217000    28
```

Filtrado de columnas

```
green_rice_W <- select(green_rice, -uptake, -N)
str(green_rice_W)
```

```
tibble [64 x 3] (S3: tbl_df/tbl/data.frame)
 $ W      : int [1:64] 10 10 10 10 40 40 40 40 0 0 ...
 $ Replicate: int [1:64] 1 1 1 1 1 1 1 1 1 1 ...
 $ Main    : int [1:64] 1 1 1 1 2 2 2 2 3 3 ...
```

Ordenación de datos

Es posible ordenar en función de los valores de una columna o de varias mediante la función `arrange()`. En el caso de que usemos varias, primero se ordena por los valores de la primera columna y, para cada uno de ellos, por los valores de la siguiente columna especificada.

8. Análisis exploratorio de datos

```
barley_byState <- arrange(nass_barley, state)
head(barley_byState)
```

```
# A tibble: 6 x 4
  year state   acres yield
  <int> <chr>   <dbl> <dbl>
1  1943 Alabama  5000   18
2  1944 Alabama  5000  20.5
3  1945 Alabama  2000   20
4  1946 Alabama  2000   18
5  1947 Alabama  1000   18
6  1948 Alabama  2000   19
```

```
barley_byYear_yield <- arrange(nass_barley, year, yield)
head(barley_byYear_yield)
```

```
# A tibble: 6 x 4
  year state   acres yield
  <int> <chr>   <dbl> <dbl>
1  1866 Texas      3000   16
2  1866 Pennsylvania 30000  18.5
3  1866 Missouri    9000   21
4  1866 Maine     34000  21.5
5  1866 Iowa     66000   22
6  1866 New Hampshire 4000   22
```

Por defecto, la ordenación se hace en sentido ascendente para valores numéricos (de menor a mayor valor) y en orden alfabético para cadenas de caracteres (datos cualitativos).

Si queremos invertir el orden de la operación de ordenación podemos usar la función `desc()`.

```
barley_byYear_desc <- arrange(nass_barley, desc(year),)
head(barley_byYear_desc)
```

```
# A tibble: 6 x 4
  year state      acres yield
  <int> <chr>      <dbl> <dbl>
1  2011 Alaska      4800  36.5
2  2011 Arizona    64000  125
3  2011 California 75000   63
4  2011 Colorado   63000  126
5  2011 Delaware   32000   88
6  2011 Idaho     500000   93
```

Selección de filas

La función `slice()` nos permite seleccionar un conjunto de filas según su índice, teniendo en cuenta que la primera fila tiene el índice 1.

```
# Recuperamos 90 observaciones, de la fila 11 a la 100
barley_10_100 <- nass_barley |>
  slice(11:100)
nrow(barley_10_100)
```

```
[1] 90
```

Tip

La función `slice()` viene acompañada de otras funciones de ayuda que implementan casos comunes: `slice_head()`, `slice_tail()`, `slice_sample()`, `slice_min()`, `slice_max()`. Para conocer más detalles consulta la página de documentación de `slice()`.

8. Análisis exploratorio de datos

8.3.2. Transformación de datos

Definición de nuevas variables

Se pueden definir nuevas columnas que representen variables adicionales mediante la función `mutate()`. Para cada nueva variable, especificamos el nombre de la columna y la operación a realizar para obtener cada valor de esa nueva columna. Se pueden utilizar los valores de otras columnas previas.

```
corn_high_surf <- mutate(nass_corn,
  harvest_index = acres/yield,
  year_factor = as.factor(year),
  is_high = acres > median(acres),
  crop = "corn",
  relative_yld = yield/mean(yield))

head(corn_high_surf)
```

```
# A tibble: 6 x 9
  year state  acres yield harvest_index year_factor is_high crop relative_yld
  <int> <chr>   <dbl> <dbl>         <dbl> <fct>      <lgl> <chr>         <dbl>
1  1866 Alaba~ 1.05e6     9         116667. 1866      TRUE  corn           0
2  1866 Arkan~ 2.8 e5     18         15556. 1866     FALSE  corn           0
3  1866 Calif~ 4.20e4     28          1500 1866     FALSE  corn           0
4  1866 Conne~ 5.7 e4     34          1676. 1866     FALSE  corn           0
5  1866 Delaw~ 2 e5      23          8696. 1866     FALSE  corn           0
6  1866 Flori~ 1.25e5     9         13889. 1866     FALSE  corn           0
```

Renombrado de columnas

Se puede cambiar el nombre de una columna con la función `rename(dataframe, newname = "oldname")`.

8.4. Resúmenes estadísticos

Separación de valores

Puede ocurrir que los datos no cumplan los criterios de *tidy data*, por ejemplo, porque existan múltiples valores en una misma celda.

8.3.3. Combinación de tablas de datos (funciones join)

8.4. Resúmenes estadísticos

8.5. Gráficos exploratorios

9. Creación de gráficos

In summary, this book has no content whatsoever.

See Knuth (1984) for additional discussion of literate programming.

10. Modelos con R

In summary, this book has no content whatsoever.

See Knuth (1984) for additional discussion of literate programming.

11. Recursos adicionales

11.0.1. Ejercicios y referencias sobre tidyverse

Llegados a este punto, lo más importante es realizar la mayor cantidad posible de ejercicios y ejemplos para afianzar nuestro conocimiento de la interfaz de `dplyr`. En este punto, sobre todo si tenemos todavía poca experiencia en programación (pero si la tenemos, *también*), es importante tener **mucha paciencia** tanto con el propio lenguaje como con nosotros mismos.

`dplyr` define una serie de operaciones para manipulación y consulta de datos que no son arbitrarias. Llevan definidas desde hace décadas, cuando expertos en bases de datos pensaron en la forma de organizar la información de manera más eficiente y cómo recuperarla y modificarla posteriormente.

Por tanto, es importante no desesperarse y poco a poco ir habituándose a las funciones existentes y a **construir consultas** con estas piezas básicas.

Recomendamos a continuación varias fuentes en las que se pueden encontrar muchos ejemplos adicionales y ejercicios para practicar:

1. La versión online del libro *R for Data Science* de H. Wickham (Wickham & Grolemund, 2017) incluye en el capítulo 5 más ejemplos de estas mismas operaciones, utilizando datos del paquete `nycflights13`.

11. Recursos adicionales

2. Los materiales en línea del curso **Stat545** (<https://stat545.com/>), creado por J. Bryan contienen varios capítulos relevantes para repasar y practicar. Varios de estos capítulos también proponen ejercicios adicionales para resolver:
 - El capítulo 6 es una introducción a las operaciones básicas con **dplyr**.
 - El capítulo 7 amplía con más ejemplos la explicación de las funciones que solo involucran a una tabla.
 - El capítulo 14 se muestra como ampliar una tabla añadiendo más filas o más columnas procedentes de otras tablas.
 - El capítulo 15 amplía las operaciones *join*, explicando gráficamente y de forma clara cómo se combinan las tablas con cada operación y el resultado final.
3. El capítulo 3 del libro *Modern Dive into R* de C. Ismay y A.Y. Kim, contiene más ejemplos y explicaciones gráficas para entender todas estas operaciones.

11.1. Recursos para investigación en Agricultura

- R Packages for Agricultural Research (j. Piaskowski, Univ. of Idaho, 2024).

Referencias

- Codd, E. F. (1990). *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc.
- Gomez, K. A., & Gomez, A. A. (1984). *Statistical Procedures for Agricultural Research*. John Wiley & Sons.
- Kelleher, J. D., Namee, B. M., & D'Arcy, A. (2015). *Fundamentals of Machine Learning for Predictive Data Analytics: Algorithms, Worked Examples, and Case Studies*. MIT Press Ltd.
- Knuth, D. E. (1984). Literate Programming. *Comput. J.*, 27(2), 97-111. <https://doi.org/10.1093/comjnl/27.2.97>
- Kuhn, M., & Johnson, K. (2013). *Applied Predictive Modeling*. Springer New York. <https://books.google.es/books?id=xYRDAAAQBAJ>
- Rodney Edmondson, Hans-Peter Piepho, & Muhammad Yaseen. (2020). *agriTutorial: Tutorial Analysis of Some Agricultural Experiments*.
- Tukey, J. W. (1977). *Exploratory Data Analysis* (1.^a ed.). Addison-Wesley Pub.
- Wickham, H. (2014). Tidy Data. *Journal of Statistical Software, Articles*, 59(10), 1-23. <https://doi.org/10.18637/jss.v059.i10>
- Wickham, H., & Grolemund, G. (2017). *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data* (1st ed.). O'Reilly Media, Inc.
- Wright, K. (2024). *agridat: Agricultural Datasets*. <https://CRAN.R-project.org/package=agridat>

A. Comandos de utilidad

A.1. Comandos en R Studio

A.2. Comandos con `dplyr`

B. Entornos de desarrollo

B.1. R Studio

B.2. Visual Studio

C. Paquetes R destacados

C.1. tidyverse

C.2. ggplot2

C.3. GGally

C.4. *Pipelines*

C.5. data.table

C.6. Modelos estadísticos y aprendizaje automático

C.6.1. tidymodels

C.6.2. mlr

Referencias

- Codd, E. F. (1990). *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc.
- Gomez, K. A., & Gomez, A. A. (1984). *Statistical Procedures for Agricultural Research*. John Wiley & Sons.
- Kelleher, J. D., Namee, B. M., & D'Arcy, A. (2015). *Fundamentals of Machine Learning for Predictive Data Analytics: Algorithms, Worked Examples, and Case Studies*. MIT Press Ltd.
- Knuth, D. E. (1984). Literate Programming. *Comput. J.*, 27(2), 97-111. <https://doi.org/10.1093/comjnl/27.2.97>
- Kuhn, M., & Johnson, K. (2013). *Applied Predictive Modeling*. Springer New York. <https://books.google.es/books?id=xYRDAAAQBAJ>
- Rodney Edmondson, Hans-Peter Piepho, & Muhammad Yaseen. (2020). *agriTutorial: Tutorial Analysis of Some Agricultural Experiments*.
- Tukey, J. W. (1977). *Exploratory Data Analysis* (1.^a ed.). Addison-Wesley Pub.
- Wickham, H. (2014). Tidy Data. *Journal of Statistical Software, Articles*, 59(10), 1-23. <https://doi.org/10.18637/jss.v059.i10>
- Wickham, H., & Grolemund, G. (2017). *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data* (1st ed.). O'Reilly Media, Inc.
- Wright, K. (2024). *agridat: Agricultural Datasets*. <https://CRAN.R-project.org/package=agridat>

