Básicos de R: lectura/escritura, manejo de datos, figuras

David García Callejas

Lectura/escritura de archivos

La estructura de datos más apropiada para cargar datos en R son, generalmente, los dataframes o las matrices. Por supuesto, R acepta muchos tipos de archivos especiales (por ejemplo, archivos raster o vectoriales para funciones SIG), pero nos centraremos en cargar y escribir datos en formato tabla.

Comenzaremos por repasar las funciones básicas de lectura y escritura de datos en R.

```
?read.table
?write.table
```

read.table es la función general para leer archivos de texto en tablas. En la función, el primer argumento que se especifica es la ruta del archivo a leer. Otros argumentos importantes son "header", que indica si el archivo incluye nombre de columnas; "sep", que indica el carácter que sirve de separación entre valores, o "dec", que indica el carácter que sirve como punto decimal.

Esta y otras funciones asociadas devuelven un dataframe al leer un archivo. En la carpeta "data" hay dos archivos exactamente iguales con extensión .csv, uno separado por comas y otro separado por punto y coma, que nos pueden servir de primer ejemplo.

```
mi.archivo <- read.table(file = here::here("data","starwars_names.csv"),
                         header = TRUE,
                         sep = ";")

head(mi.archivo)
str(mi.archivo)
```

Existen funciones pensadas para facilitar (un poco) la vida, que tienen opciones por defecto para dos tipos de caracteres de separación. Para leer directamente un archivo csv separado por comas, podemos usar la funcion read.csv

```
a2 <- read.csv(file = here::here("data","starwars_names_coma.csv"))
```

y para leer csv separado por punto y coma, la función read.csv2

```
a3 <- read.csv2(file = here::here("data","starwars_names.csv"))
```

Estas funciones también asignan valores por defecto al carácter de separación decimal, tened cuidado.

R también permite leer hojas de excel como dataframes. Para ello necesitamos cargar un *paquete* externo. Un paquete no es más que una colección de funciones que no están incluidas por defecto en nuestra instalación de R. En este caso, por defecto R no trae funciones para leer archivos de excel, pero la comunidad ha desarrollado estas funciones y nosotros podemos cargarlas y usarlas.

```
# la orden "library" se usa para cargar un paquete determinado
library(readxl)
hoja1 <- read_excel(path = here::here("data","hojas_excel.xlsx"),sheet = "parcela_1")
```

Existen funciones específicas que sirven para leer de manera más eficiente archivos csv muy grandes, de miles o millones de filas. Nosotros no trabajaremos con archivos tan grandes, pero estas funciones están disponibles en el paquete "readr", y son practicamente iguales a las funciones por defecto de R.

```
library(readr)
?read_csv2
```

¿Cómo sabemos la ruta hacia un archivo? Si os habéis fijado, yo he usado rutas relativas. Para entender esto necesitamos saber lo que es el directorio de trabajo en R.

```
getwd()
```

Una sesión de R siempre trabaja teniendo una carpeta (o directorio) como base. Esta carpeta es aquella en la que, si no especificamos otra cosa, se guardan o se leen los datos que nos interesen. Al abrir una sesión nueva de R/Rstudio, nuestro directorio de trabajo será el que R asigna por defecto. en Linux, el directorio base del usuario. ¿y en Windows? abrid una sesión nueva en Rstudio y llamad a la función getwd.

Este directorio de trabajo se puede modificar con la función setwd

```
?setwd
```

En todo caso, si conocemos nuestro directorio de trabajo, podemos usar rutas relativas para leer nuestros archivos. Los dos puntos de más arriba "../" suben un nivel en la estructura de carpetas. Desde ese nivel, dentro de la carpeta "data", ya encuentro, en mi equipo, los archivos que me interesan. Para ayudar a la búsqueda, igual que con variables/funciones, puedo usar el tabulador.

Las funciones de lectura tienen sus equivalentes funciones de escritura, write.table y las asociadas. En este caso tenemos que especificar 1) el dataframe/matriz que queremos escribir, y 2) la ruta y el nombre con el que queremos guardar el archivo.

```
?write.table

a2$name[1] <- "Lucas"

# guardar archivo csv separado por punto y coma
# write.table es la función general
# podemos especificar si queremos una columna que guarde
# los nombres de las filas de nuestro dataframe

# estas dos órdenes son equivalentes
# write.table(x = a2,file = here::here("data","archivo_nuevo.csv"),
#             sep = ";",
#             row.names = FALSE)
# write.csv2(x = a2,file = here::here("data","archivo_nuevo.csv"),
#            row.names = FALSE)

# y para guardar un archivo csv separado por comas, usamos
# write.csv(x = a2, file = here::here("data","archivo_nuevo_comas.csv"),
#           row.names = FALSE)
```

Excel puede leer perfectamente los archivos .csv, y son mucho más fáciles de compartir entre sistemas operativos, programas de hojas de cálculo y otro software. Os recomiendo guardar *siempre* vuestras tablas en formato .csv (o .txt). En todo caso, existen paquetes de R que permiten guardar archivos directamente en formato excel.

```
# install.packages("xlsx")
# library(xlsx)
# ?write_xlsx
```

Manejo de datos

En esta sección repasaremos el manejo de datos de diferentes tipos. La selección, modificación, eliminación, o agregación de datos, principalmente usando dataframes.

Selección

```
eq <- read.csv2(file = here::here("data","Earthquake_data.csv"),
                header = TRUE,
                dec = ".",
                stringsAsFactors = FALSE)

# selección de columnas
eq.clean <- eq[,c("En","Year","Mo","Da","Ho",
                  "Mi","Se","Area","Lat","Lon",
                  "Dep","M")]

# selección de observaciones. P.ej terremotos de más de 7 grados de magnitud
eq.7 <- subset(eq.clean,M > 7)
summary(eq.7$M)

# si en vez de recuperar los datos completos sólo me interesa
# la posición de las observaciones con magnitud >7, uso which
pos.7 <- which(eq.clean$M > 7)

# y si quiero recuperar el dataframe sólo con estas posiciones:
eq.7.2 <- eq.clean[pos.7,]

# la manera de seleccionar con subset y con which
# debería darnos dataframes equivalentes
identical(eq.7,eq.7.2)

# si tenemos filas duplicadas, podemos eliminarlas con la función unique
# en este caso, no existen duplicados,
# así que la función devuelve un dataframe igual al original
eq.7.3 <- unique(eq.7)
```

Hay situaciones en las que una variable puede tener valor nulo (NULL). De manera similar, una observación en un dataframe o matriz (es decir, una fila) puede no tener un valor asociado en alguna de las columnas. Por ejemplo, puede haber terremotos de los que no sepamos con seguridad la hora a la que sucedieron.

```
# columna Ho indica la hora, Mi los minutos, Se los segundos
head(eq.clean)
```

Estos valores de los que no tenemos datos se indican con la palabra reservada "NA". Algunas de las operaciones más comunes en dataframes implican limpiar los NAs de nuestros conjuntos de datos, por ejemplo para realizar análisis estadísticos. Las funciones que hemos visto para seleccionar datos también nos sirven para esto.

```
# dejar sólo observaciones en las que no haya ni un solo NA en ninguna columna
eq.sin.na.vec <- complete.cases(eq.clean)

# fijaos que la función devuelve un vector de valores lógicos,
# es decir, un valor TRUE o FALSE por cada fila de nuestros datos.
# Para seleccionar estos datos, lo hacemos como antes con "which"
eq.sin.na <- eq.clean[eq.sin.na.vec,]
eq.sin.na

# estas dos operaciones se pueden realizar en una sola línea
eq.sin.na <- eq.clean[complete.cases(eq.clean),]
```

```r
# podemos seleccionar también los valores de una sola columna
# por ejemplo, aquellos terremotos de los que sepamos la profundidad
# estas dos maneras son equivalentes:
# en subset, usamos el nombre de la columna directamente
eq.profundidad <- subset(eq.clean, !is.na(Dep))
# en which, necesitamos especificar dataframe$columna
eq.profundidad.2 <- eq.clean[which(!is.na(eq.clean$Dep)),]

# hemos usado la función is.na, cuyo objetivo es tan simple
# como decirnos qué elementos de un vector son NA.
ejemplo.vector <- c("este", "vector", "tiene", "algún", "NA", NA)

is.na(ejemplo.vector)
which(is.na(ejemplo.vector))
# recordad que la exclamación delante de una condición lógica es
# la negación de esa condición
which(!is.na(ejemplo.vector))
# ¿cuántos NA hay en un vector?
# la función "sum" también funciona con vectores lógicos
sum(is.na(ejemplo.vector))
```

Modificación

Todas las funciones anteriores, en las que seleccionamos filas determinadas de un dataframe, nos pueden servir para modificar los valores que queramos

```r
# no es recomendable en general, pero podemos
# sustituir los NAs por otros valores.
# primero copiamos el dataframe original, para evitar pisar datos
eq.sust <- eq.clean

# asignamos 0 a aquellas horas que sean NA
eq.sust$Ho[which(is.na(eq.sust$Ho))] <- 0

# ¿cómo son ahora los datos de hora?
summary(eq.clean$Ho) # originales
summary(eq.sust$Ho) # con 0 en los NAs

# también podemos sustituir valores de cualquier otro tipo, no sólo NA
# por ejemplo, el campo "Area" es bastante confuso. Nos puede interesar
# aislar los terremotos en dos regiones. Por ejemplo, Chile por un lado y
# el resto del mundo por otro

head(eq.sust$Area)

eq.sust$Area[which(eq.sust$Area != "[Chile]")] <- "other"
head(eq.sust$Area)
table(eq.sust$Area)
```

Aquí no las trataremos en detalle, pero existen maneras de seleccionar caracteres más sofisticadas, por medio de las denominadas “expresiones regulares”, o “regular expressions” (regex) en inglés. Son una serie de reglas que podemos aplicar para seleccionar qué elementos de un vector de caracteres cumplen un determinado patrón, o sustituir unos caracteres por otros.

```r
# podemos sustituir un carácter dentro de los elementos de un vector
# por cualquier otro carácter
# por ejemplo, espacios por guiones
area.orig <- eq.clean$Area

# la función gsub reemplaza un patrón de caracteres por otro
area.guiones <- gsub(pattern = " ", replacement = "_",x = area.orig)
```

Agregación

Es muy común tener conjuntos de datos dispersos en varias tablas. Por ejemplo, podríamos tener los datos de terremotos en diferentes tablas, una de las cuales fueran las características físicas de la zona, otra la descripción del terremoto, otra tabla para daños causados, etc. R nos permite unir información de diferentes tablas, y hay varias formas para ello.

```r
# match es una función muy útil
# nos indica qué valores de un vector se corresponden
# con los valores de un segundo vector
v1 <- c(1,2,3,4,5)
v2 <- c(8,4,9,7,0,1)

pos <- match(v1,v2)
pos

# el primer elemento de v1 está en la posición 6 de v2.
# El segundo y tercero no están (NA), el cuarto
# está en la posición 2 de v2, y el quinto no está.

pos2 <- match(v2,v1)
pos2

# match nos puede servir para unir dataframes en base a una columna común
d1 <- data.frame(caracter = c("a","b","c"), num = c(1,2,3))
d1
d2 <- data.frame(caracter = c("c","d","e"), mas.info = c(6.6,3.4,8.7))
d2

# añadimos la columna mas.info de d2 en d1, en base a la columna caracter que comparten.
d1$mas.info <- d2$mas.info[match(d1$caracter,d2$caracter)]
d1

# como d1 y d2 sólo comparten el caracter c,
# el resto de valores de la columna mas.info se asignan NA.

# otra manera muy importante y más general de unir dataframes
# en base a columnas comunes es la familia de funciones "join".
# Para usarlas, necesitamos cargar los paquetes "tidyverse",
# que trabajaremos bastante en el resto de sesiones.
library(tidyverse)
?join

# existen varias funciones para unir tablas,
# dependiendo de qué tipo de unión necesitamos.
```

```r
# Por ahora veremos un ejemplo de left_join
# usaremos dos tablas con información sobre los
# personajes de la saga de Star Wars.
# En una tabla tenemos datos físicos de cada personaje,
# y en otra tabla tenemos información sobre las naves espaciales
# que cada personaje pilota.

personajes_SW <- read.csv2(here::here("data","starwars_info_personajes.csv"),
                           header = TRUE,
                           stringsAsFactors = FALSE)
naves_SW <- read.csv2(here::here("data","starwars_personajes_naves.csv"),
                      header = TRUE,
                      stringsAsFactors = FALSE)

head(personajes_SW)
head(naves_SW)

# podemos unir ambas tablas usando como columna común
# el nombre de los personajes.
SW_conjunta <- left_join(x = personajes_SW,y = naves_SW, by = "name")

# left_join añade las columnas de y (el segundo dataframe) a x (el primero).
head(SW_conjunta)
# en este caso todas las filas tienen su correspondencia en x e y
# (todos los nombres de x están en y, y viceversa).
# cuando esto no sucede, left_join devuelve valores NA donde corresponda.
```

Una cuestión importante en cuanto a la estructura de los dataframes es entender cómo están organizados los datos y qué implicaciones tiene esa organización para los análisis posteriores. En particular, hay dos maneras generales de organizar observaciones en dataframes (ver, por ejemplo, https://sejdemyr.github.io/r-tutorials/basics/wide-and-long/). Una es el formato "wide" (ancho), en el que cada variable o factor es una columna del dataframe.

```r
# en estos datos, cada nave es una columna
head(naves_SW)

# y en otro ejemplo con un subconjunto de terremotos,
# podemos tener la magnitud de cada terremoto en cada Área ordenada por años
eq.wide.data <- read.csv2(here::here("data","Earthquake_wide_example.csv"),
                          header = TRUE)
head(eq.wide.data)
# fijaos que R añade una "X" al principio de cada año/columna.
# Esto es porque nombres de columnas que empiecen por un número
# son considerados incorrectos.
# Si queréis evitar esto, usad el argumento "check.names = FALSE"
# al leer los datos.
```

El formato "long" (largo) agrupa las variables o factores en una columna y los valores en otra. Aunque pueda parecer contraintuitivo, es mucho más eficiente en general trabajar con datos en formato long. En nuestro ejemplo de Star Wars, es mucho más sencillo comparar visualmente los diferentes vehículos para un mismo personaje. Además, varias funciones avanzadas de R y otros lenguajes de programación trabajan con datos long.

```r
# podemos transformar datos de wide a long con las funciones "pivot"
# especificando
```

```r
# qué columnas queremos incluir (cols), por su número,
# qué nombre tendrá la columna que diferencia los factores (names_to)
# qué nombre tendrá la columna de los datos en sí (values_to)
naves_SW_long <- pivot_longer(data = naves_SW,
                              cols = 2:17,
                              names_to = "spaceship",
                              values_to = "is.pilot")
head(naves_SW_long)

# otro ejemplo, con los datos simplificados de terremotos
eq_long <- pivot_longer(data = eq.wide.data,
                        cols = 2:47,
                        names_to = "year",
                        values_to = "magnitude")
head(eq_long)
# en este caso hay muchas combinaciones de área y año
# que no tuvieron terremotos, por lo que tenemos muchos NAs.
# podemos eliminarlos como hicimos más arriba
eq_long_clean <- eq_long[which(!is.na(eq_long$magnitude)),]
head(eq_long_clean)
```

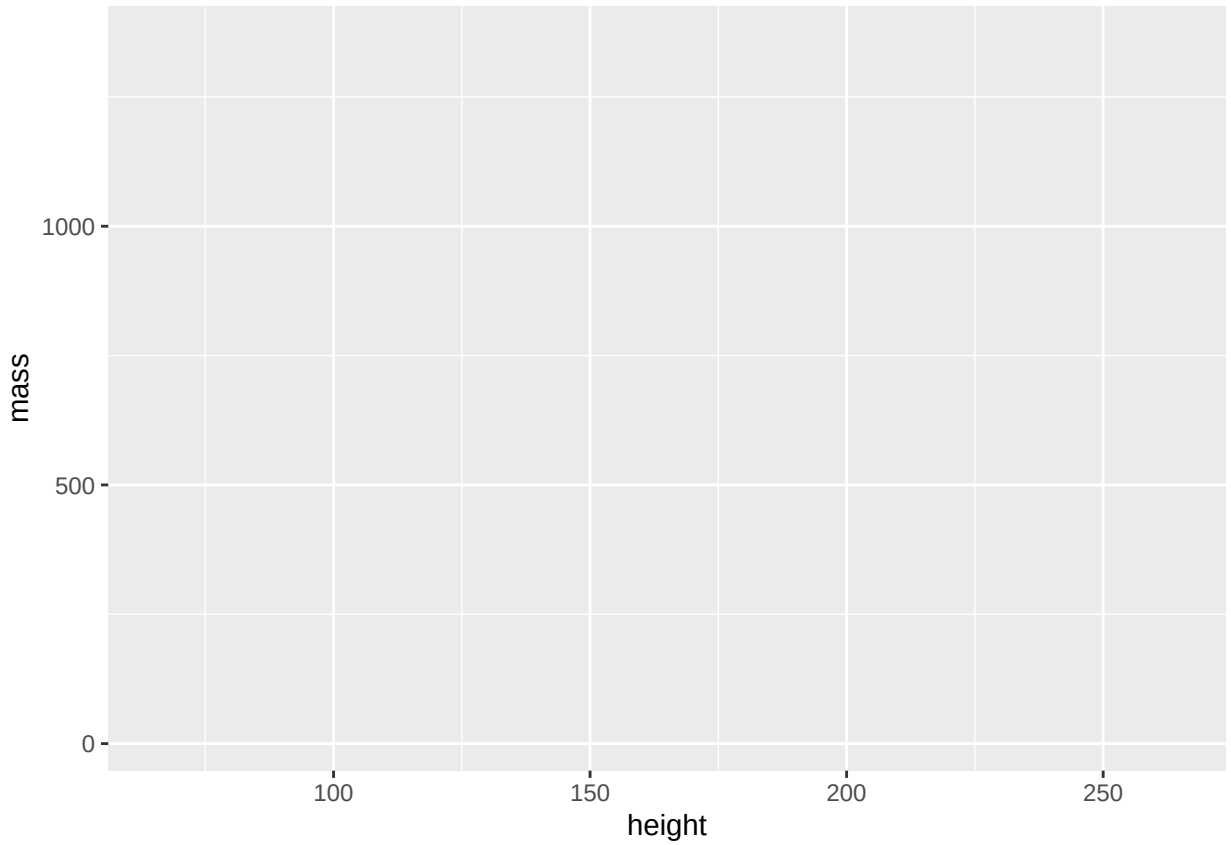
Figuras con ggplot2

Para crear figuras en R usaremos el paquete `ggplot2`, que es seguramente el más usado hoy día para producir gráficos de calidad. R tiene otros paquetes y funciones para generar gráficos, pero `ggplot2` es intuitivo, fácilmente editable, y tiene una documentación muy buena. El acrónimo `ggplot2` viene de "grammar of graphics plots", por el hecho de que este conjunto de funciones esta basado en la llamada "Grammar of Graphics", una sintaxis que estructura la creación de gráficos a partir de capas con diferente información.

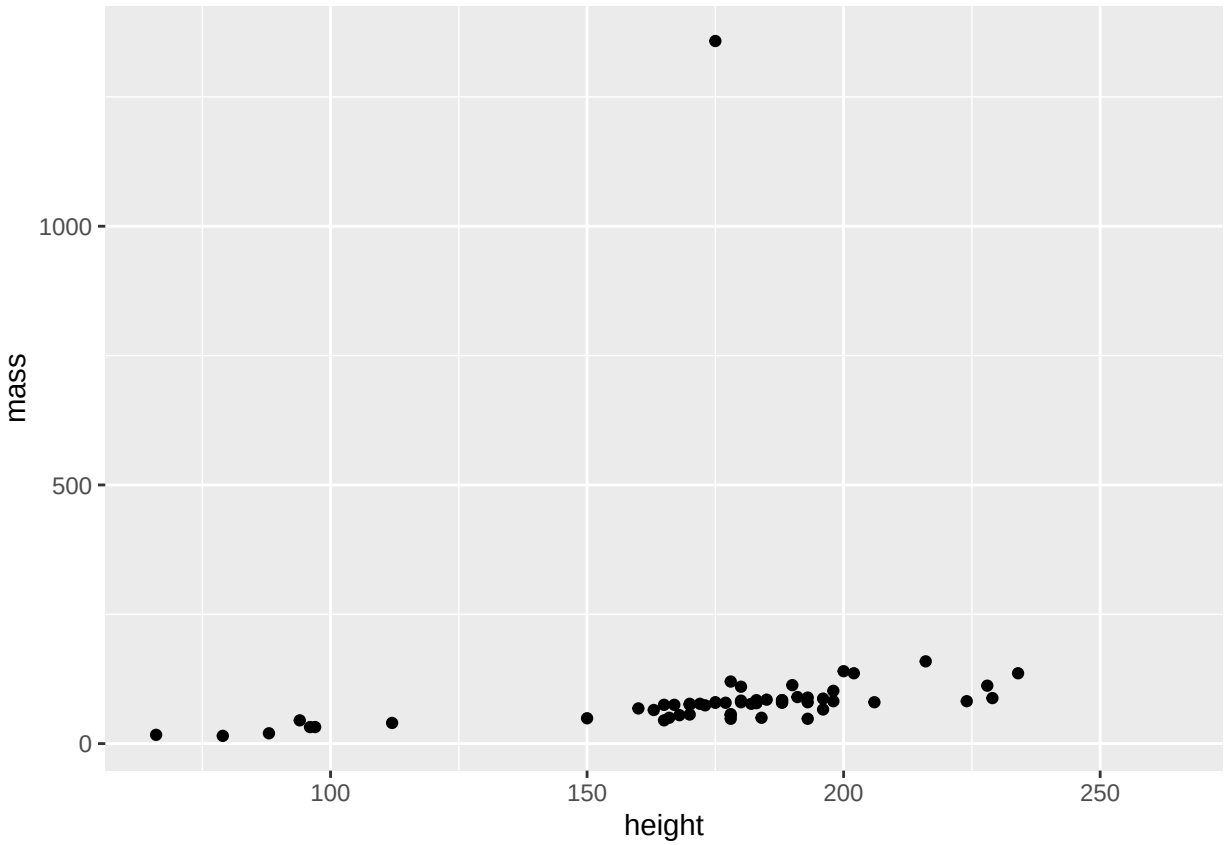
Como primer ejemplo, vamos a crear un gráfico de puntos.

```r
# este paquete incluye ggplot2
library(tidyverse)

# leemos los datos de personajes de Star Wars
personajes_SW <- read.csv2(file = here::here("data",
                                             "starwars_info_personajes.csv"),
                           stringsAsFactors = FALSE)

# queremos crear una figura que muestre la altura y el peso de cada personaje,
# donde cada personaje sea un punto

# lo primero que tenemos que entender es que un "plot", una figura,
# también la podemos guardar en una variable.
# Para crear una figura nueva, usamos la función ggplot
# en esta función, tenemos que especificar qué datos
# queremos usar para la figura

figura1 <- ggplot(data = personajes_SW)

# ¿qué contenido tiene este objeto?
figura1 # en Rstudio, aparecerá la zona de "plots" en blanco.
```

```r
class(figura1) # este objeto es de clase ggplot

# a partir de aquí, podemos ir añadiendo "capas" de información
# a nuestra figura vacía
# lo primero que debemos pensar en incluir son los ejes de nuestra figura.
# Esto se puede hacer también en la función ggplot

# el argumento "aes" significa "aesthetics", es decir,
# la estética, los detalles de la figura.
figura1 <- ggplot(data = personajes_SW,aes(x = height, y = mass))
figura1
```




```
# ahora deberíamos ver los ejes sobre los que queremos dibujar nuestros puntos.
# Estos puntos, uno por cada personaje, serán una capa nueva de información.
# Las capas se añaden en ggplot2 con las funciones "geom_",
# literalmente añadiendo elementos al objeto que hemos creado
figura1 <- figura1 + geom_point()
figura1
```



Con esto ya tenemos nuestro primer gráfico con ggplot2. Podemos guardar cualquier gráfico que creemos de diferentes maneras. Por ejemplo, Rstudio nos permite exportar cualquier gráfico que aparezca en la pestaña "Plots", usando el botón "Export". Sin embargo, es más recomendable usar la función `ggsave` del propio paquete ggplot2, que da más control y automatización.

```
# ggsave por defecto usa pulgadas como unidades...
# yo suelo ir haciendo ensayo-error con diferentes tamaños
# hasta encontrar el balance adecuado entre tamaño de letra/puntos, y fondo
ggsave(filename = "mi_figura.png",plot = figura1,width = 5,height = 5)
```

Antes de ver otros tipos de figuras, veremos cómo retocar esta figura inicial. Una primera mejora puede ser colorear cada punto según su especie.

```
# lo primero que haremos es, como antes, eliminar a Jabba,
# que nos descuadra toda la imagen.
sw2 <- subset(personajes_SW, name != "Jabba Desilijic Tiure")

# y creamos una figura en la que cada especie tenga un color diferente
# para esto añadimos un "aesthetic" más,
# pero dentro de la función que dibuja los puntos.
figura2 <- ggplot(sw2, aes(x = height, y = mass)) +
  geom_point(aes(color = species))

# la figura funciona, pero hay muchas,
# demasiadas especies como para que sea efectiva.
# podemos agrupar las especies en grupos más grandes, para simplificar.
# ¿Cuántos personajes hay de cada especie?
table(sw2$species)
```

```
# de casi todas las especies hay un solo individuo.
# Dejaremos los droides, los humanos,
# y, por curiosidad, a Yoda. El resto los agruparemos en una categoría "otros"
# primero, creamos una columna idéntica a "species"
sw2$species_group <- sw2$species
# y agrupamos aquellas que no son ni "Human", ni "Droid", ni "Yoda's species"
sw2$species_group[which(!sw2$species %in% c("Human",
                                            "Droid",
                                            "Yoda's species"))] <- "Other"
table(sw2$species_group)

# rehacemos nuestra figura con la nueva clasificación
figura2 <- ggplot(sw2, aes(x = height, y = mass)) +
  geom_point(aes(color = species_group))

# también podemos asignar valores fijos a todos los puntos.
# Fijaos en la diferencia entre la orden anterior y esta
figura2.2 <- ggplot(sw2, aes(x = height, y = mass)) +
  geom_point(color = "darkgreen")
```

Al incluir "color" o cualquier otro argumento dentro de `aes`, estamos diciendo que queremos que el color varíe con respecto a otra columna, en este caso "species_group". Pero si dentro de `geom_point` especificamos directamente un color, todos los puntos quedarán cambiados. Los atributos más comunes que podemos modificar son, además del color, la forma (shape), tamaño (size), o el relleno (fill).

```
figura2.3 <- ggplot(sw2, aes(x = height, y = mass)) +
  geom_point(color = "darkgreen", size = 3, aes(shape = species_group))
```

Las formas que pueden tener los puntos en R vienen dadas por un código numérico. Aquí podéis ver a qué forma corresponde cada código: http://www.cookbook-r.com/Graphs/Shapes_and_line_types/

Al crear la figura, ggplot elimina automaticamente los datos que sean NA. Al lanzar el código en vuestro ordenador, habréis visto un "warning" avisando de que hay 28 filas con NA. Antes de continuar explorando otras opciones para retocar la leyenda, las etiquetas, títulos, ejes, etc, veremos otros tipos de figuras que podemos crear.

```
# gráfico de barras: cuántos individuos hay por cada grupo de especies.
# Para esta figura no necesitamos añadir eje y,
# porque este será el conteo de cada grupo.
# el conteo se calcula con geom_bar, añadiendo el argumento "stat = "count""
figura3 <- ggplot(sw2, aes(x = species_group)) +
  geom_bar(stat="count")

# como antes, podemos colorear cada grupo.
# Para colorear areas, usamos la orden "fill",
# que significa, literalmente, "relleno".
figura3 <- ggplot(sw2, aes(x = species_group)) +
  geom_bar(stat="count", aes(fill = species_group))

# gráfico de cajas (boxplot)
figura4 <- ggplot(sw2, aes(x = species_group, y = height)) +
  geom_boxplot(aes(fill = species_group))

# distribución de puntos
figura5 <- ggplot(sw2, aes(x = species_group, y = height)) +
```

```
  geom_jitter(aes(color = species_group))

# puntos y cajas
figura6 <- ggplot(sw2, aes(x = species_group, y = height)) +
  geom_jitter(aes(color = species_group), alpha = .8) +
  geom_boxplot(aes(fill = species_group), alpha = .5)

# histogramas
figura7 <- ggplot(sw2, aes(x = height)) +
  geom_histogram(bins = 30)

figura8 <- ggplot(sw2, aes(x = height)) +
  geom_density()
```

También podemos añadir regresiones a nuestras figuras. Para ello usamos la orden `geom_smooth`. Este ejemplo también nos sirve para comprobar, de nuevo, que podemos "apilar" todas las capas que queramos.

```
# Por defecto usa un método de regresión polinomial (loess)
figura9 <- ggplot(sw2, aes(x = height, y = mass)) +
  geom_point(aes(color = species_group)) +
  geom_smooth()

# pero podemos especificar otros modelos
figura10 <- ggplot(sw2, aes(x = height, y = mass)) +
  geom_point(aes(color = species_group)) +
  geom_smooth(method = "lm")

# igual que dividir el color por grupos, podemos dibujar
# una regresión por cada grupo
figura11 <- ggplot(sw2, aes(x = height, y = mass)) +
  geom_point(aes(color = species_group)) +
  geom_smooth(method = "lm", aes(color = species_group))

# algunos detalles
figura12 <- ggplot(sw2, aes(x = height, y = mass)) +
  geom_point(aes(color = species_group)) +
  geom_smooth(method = "lm",
              aes(color = species_group),se = FALSE)
```

Ahora podemos entender mejor por qué ggplot2 trabaja con datos en formato "long". En este formato, los "grupos" de datos están en una columna, en vez de divididos en varias columnas.

```
head(sw2)

sw2.ejemplo <- sw2[,c("name","height","mass","species_group")]
sw2.ejemplo$valor <- TRUE

sw3.ejemplo <- pivot_wider(data = sw2.ejemplo,
                           id_cols = c(name,height,mass),
                           names_from = species_group,
                           values_from = valor,
                           values_fill = list(valor = FALSE))
head(sw3.ejemplo)
```

Al dividir la especie en columnas, sería imposible crear una figura con todas las categorías de manera sencilla:

¿cuál sería la columna para especificar la agrupación?

```
sw3.plot <- ggplot(sw3.ejemplo, aes(x = height, y = mass)) +
  geom_point(aes(color = Human)) # esto sólo nos diferencia humanos de otros
```

Para hacer gráficos de líneas usamos los datos de terremotos. Vamos a dibujar el número de terremotos por siglo. Este ejemplo es un poco más complejo por la preparación previa que hacemos de los datos.

```
eq <- read.csv2(file = here::here("data","Earthquake_data.csv"),
                dec = ".",stringsAsFactors = FALSE)

# necesitamos crear una columna "century"
# en principio vacía
eq$century <- NA

# hay varias maneras de traducir el año a siglo.
# Por lo que hemos visto hasta ahora, podemos hacerlo con un bucle,
# que vaya fila por fila,
# calcule a qué siglo corresponde el año,
# y rellene el valor de cada terremoto.

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

  my.century <- NA

  if(eq$Year[i] < 1500){
    my.century <- "< XVI"
  }else if(eq$Year[i] >= 1500 & eq$Year[i] < 1600){
    my.century <- "XVI"
  }else if(eq$Year[i] >= 1600 & eq$Year[i] < 1700){
    my.century <- "XVII"
  }else if(eq$Year[i] >= 1700 & eq$Year[i] < 1800){
    my.century <- "XVIII"
  }else if(eq$Year[i] >= 1800 & eq$Year[i] < 1900){
    my.century <- "XIX"
  }else if(eq$Year[i] >= 1900){
    my.century <- "XX"
  }

  eq$century[i] <- my.century

}

table(eq$century)

# lo más claro es crear un segundo dataframe sólo con esta información
terremotos_siglo <- data.frame(terremotos = table(eq$century))
# al usar el resultado de "table" para crear un dataframe,
# generamos dos columnas,
# una con el nombre del siglo y otra con el número de terremotos
terremotos_siglo
# los nombres del dataframe no son del todo correctos
names(terremotos_siglo) <- c("siglo","terremotos")
# y el orden no es adecuado, no debería ser alfabético.
# En este caso, necesitamos un factor
```

```
terremotos_siglo$siglo <- factor(terremotos_siglo$siglo,
                                 levels = c("< XVI",
                                            "XVI",
                                            "XVII",
                                            "XVIII",
                                            "XIX",
                                            "XX"))
```

Una vez tenemos los datos agrupados como nos interesa, creamos la figura

```
# a veces, R tiene comportamientos extraños.
# Si no entendéis porqué hace falta "group = 1",
# no os preocupéis... yo tampoco.
# En el "cookbook" (enlace abajo), dice:
# For line graphs, the data points must be grouped so that it knows
# which points to connect.
# In this case, it is simple - all points should be connected, so group=1.
figura13 <- ggplot(terremotos_siglo, aes(x = siglo,
                                         y = terremotos,
                                         group = 1)) +
  geom_line()
```

Como antes con los puntos, podemos especificar diferentes parámetros de la línea (de nuevo, ver http://www.cookbook-r.com/Graphs/Shapes_and_line_types/)

```
figura14 <- ggplot(terremotos_siglo, aes(x = siglo,
                                         y = terremotos,
                                         group = 1)) +
  geom_line(linetype = "dashed", size = 3, color = "darkred")
```

A veces, en vez de colorear puntos de diferente color, podemos pensar en una figura diferente para cada categoría de datos. Por ejemplo, volviendo a Star Wars

```
figura15 <- ggplot(sw2, aes(x = height, y = mass))+
  geom_point() +
  geom_smooth(method = "lm") +
  facet_grid(species_group ~ .)
```

La capa "facet_grid" divide nuestra figura en función a la columna o columnas que queramos, tanto en el eje vertical como horizontal. La sintaxis es "división_vertical ~ división_horizontal". Si sólo queremos un eje de división, como en el ejemplo de arriba, usamos un punto en el otro eje (species_group ~ .)

```
figura16 <- ggplot(sw2, aes(x = height, y = mass))+
  geom_point() +
  facet_grid(gender~species_group)
```

Estas divisiones se pueden combinar con otros atributos como colores, etc.

```
figura17 <- ggplot(sw2, aes(x = height, y = mass))+
  geom_point(aes(color = species_group, shape = gender)) +
  facet_grid(gender~species_group)
```

En este último ejemplo vemos cómo, por cada atributo que incluimos en "aes", se crea una leyenda nueva. En este caso, os habréis dado cuenta de que las leyendas son reiterativas, porque la información de cada categoría ya está en las etiquetas horizontales y verticales. Pero como ejemplo sirve. Desde esta base, vamos a retocar un poco la figura mejorando las leyendas, el texto de los ejes, los colores que usamos para cada categoría, y el título de la figura.

```r
# primero, eliminamos los NA para mejorar la visualización
sw3 <- subset(sw2, !is.na(height) & !is.na(mass) & !is.na(gender))

figura18 <- ggplot(sw3, aes(x = height, y = mass))+
  geom_point(aes(color = species_group, shape = gender), size = 2) +
  facet_grid(gender ~ species_group) +
  # una nueva capa que especifica título y, opcionalmente, subtítulo
  ggtitle(label = "Peso y altura de personajes de Star Wars",
          subtitle = "según especie y género") +
  # otra capa en la que especificamos el texto de los ejes
  labs(x = "altura (cm)", y = "peso (kg)") +
  # leyendas
  # color
  scale_color_manual(name = "Tipo de especie",
                     labels = c("androide","humano", "otro", "Yoda"),
                     values = c("grey30","darkorange",
                                "darkred","darkgreen")) +
  # shape (forma)
  scale_shape_manual(name = "Género",
                     labels = c("Femenino", "Masculino", "Otro/ninguno"),
                     values = c(1,5,6))
```

Las capas que especifican los atributos vienen especificadas por la sintaxis `scale_atributo_tipo`. En este ejemplo, modificamos el atributo `color` y el atributo `shape`, y lo que hacemos es darle valores manuales, de ahí el tipo `manual`. Especificamos el nombre (name) y las etiquetas (labels) de cada valor de la leyenda, y los valores que queremos que tomen (values). Los colores se pueden especificar por su código hexadecimal, o muchos de ellos, por un nombre estandarizado (ver http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf).

Hay muchas mas opciones que se pueden modificar. La apariencia general de las figuras (los colores de fondo, las cuadrículas, etc) vienen dadas por lo que se llama "temas". En ggplot2 hay una serie de temas ya estructurados, que se pueden usar fácilmente (ver ejemplos de los temas por defecto en https://ggplot2.tidyverse.org/reference/ggtheme.html)

```r
figura19 <- ggplot(sw3, aes(x = height, y = mass))+
  geom_point(aes(color = species_group, shape = gender), size = 2) +
  facet_grid(gender ~ species_group) +
  # una nueva capa que especifica título y, opcionalmente, subtítulo
  ggtitle(label = "Peso y altura de personajes de Star Wars",
          subtitle = "según especie y género") +
  # otra capa en la que especificamos el texto de los ejes
  labs(x = "altura (cm)", y = "peso (kg)") +
  # leyendas
  # color
  scale_color_manual(name = "Tipo de especie",
                     labels = c("androide","humano", "otro", "Yoda"),
                     values = c("grey30","darkorange","darkred","darkgreen")) +
  # shape (forma)
  scale_shape_manual(name = "Género",
                     labels = c("Femenino", "Masculino", "Otro/ninguno"),
                     values = c(1,5,6)) +
  # cambiamos al tema "bw", black and white
  theme_bw()
```

Además de dividir una figura por categorías, R nos permite juntar varias figuras en una sola imagen. Para ello instalamos y cargamos el paquete `patchwork`.

```r
install.packages("patchwork")
library(patchwork)

figura.combinada <- figura9 + figura10
```

Fijaos en la diferencia entre el tema estándar y el bw.