

Notas clase 3

[1. Algunas notas para importar bases](#)

[2. Nuevas funciones tidyverse: filter, group_by, summarise, mutate, pivot_*, *_join](#)

[3. Misceláneas: De todo un poco](#)

1. Algunas notas para importar bases

Códigos actualizados para la importación de las bases de trabajo

En esta semana actualizamos el archivo para bajar las bases (se puede bajar de este [link](#)). El código para importar la base es:

```
library(data.table)
base_covid <- fread("entradas/base_covid_muestra.txt")
```

En algunos casos hubo problemas cuando se intentó importar (sobre todo en algunas computadoras con sistema operativo *mac*). El problema era básicamente con el *encoding* (cómo traduce R los caracteres especiales del español, como los acentos o la ñ), lo cual afectaba el nombre de las variables y también su contenido. En caso de haber tenido problemas con el código anterior este código permite solucionarlo:

```
base_covid<- read.table(archivo, sep = "", header = T,
                        fileEncoding = "WINDOWS-1252")
```

Resumen de funciones de importación

Algunas funciones para importar archivos ordenados según el tipo de archivo.

```
### Archivos de texto plano (con R Base)
read.table() # --> .txt (de texto)
read.csv()   # --> .csv (separados por coma)
read.csv2()  # --> .csv (separados por punto y coma y coma
como decimal)
```

```
### Archivos de texto plano (con paquetes)
```

```

library(data.table)
fread()           # Identifica de modo automático el formato

library(readr)    # Es parte de tidyverse
read_csv()

### Archivos EXCEL
library(openxlsx)
library(readxl)
library(readr)


read_xlsx
read.xlsx()       # --> .xlsx (excel)
read_excel(path = "")

### softwares estadísticos
library(haven)
read_spss()       # .sav
read_stata()      # .dta
read_sas()        # .sas

```

Importación de bases usando los asistentes

En RStudio hay otras dos maneras de realizar las importaciones sin usar directamente los códigos. Estas pueden ser útil para ayudarnos cuando nos olvidamos un código.

La primera de ellas es usando la pestaña *Environment* (habitualmente arriba a la derecha), usando el ícono . Allí podemos elegir la fuente desde la cual queremos importar nuestro archivo y se abrirá un asistente para ayudarnos con la importación, donde podremos definir algunos parámetros de nuestra importación. Una vez que aceptemos se ejecutará el código.

Una segunda manera es en la pestaña *File* (habitualmente abajo a la derecha), que nos permite hacer una exploración de archivos y carpetas de nuestra computadora. Dentro de esta pestaña podemos presionar sobre algunos archivos que tienen extensiones conocidas (por ejemplo “.csv”) y RStudio nos propondrá “Import Dataset”, lo cual nos abrirá una nueva ventana de interacción, que nos permite definir los parámetros. En esta ventana existe la

posibilidad de copiar el código resultante sin ejecutar la importación, para incorporar directamente a nuestro script.

Es importante tener presente que siempre R ejecuta el código para hacer las importaciones, aunque usemos estas formas interactivas. Por ello, es buena idea copiar estos códigos y sumarlos a nuestro script (evitando que nuestro proceso se vuelva manual y, por lo tanto, no reproducible).

Script para importar varias bases a la vez

```
# Primero cargaremos algunas librerías:
library(tidyverse)

# Supongamos que tenemos nuestros archivos con las bases
# dentro de la carpeta "entradas/bases", la cual
# está dentro del directorio de trabajo donde tenemos
# nuestro proyecto. Es importante que los archivos
# tengan una estructura de columnas idéntica entre sí,
# con los mismos nombres y con la misma cantidad de
# columnas.

# Primero definimos la ruta donde estarán nuestros archivos
# dentro del proyecto.
ruta <- "entradas/bases"

# Generamos un vector character con el nombre de los archivos
# que hay en esta ruta y le agregamos la ruta
archivo <- list.files(ruta)
archivo <- paste0(ruta, "/", archivo)

# Este comando no lo hemos visto en clase, pero permite
# aplicar a cada ruta del vector "archivo" la función que
# le damos como segundo argumento. En este caso usamos
# read.csv(), pero puede variar según qué tipo de archivo
# queremos importar.
base <- lapply(archivo, read.csv) %>%
  bind_rows()
```

2. Nuevas funciones tidyverse: filter, group_by, summarise, mutate, pivot_*, *_join

Resumen de los operadores lógicos

Muy útiles en filter, pero no solo...

Condicionales		
Operador	Función	Método ASCII
==	Igual que	alt + 61 (=)
!=	Distinto que	alt + 33 (!)
&	Y	alt + 38
	O	alt + 124
>=	Igual o mayor que	alt + 62 (>)
<=	Igual o menor que	alt + 60 (<)
%in%	Incluido en	alt + 37 (%)

Uso de operadores para filtrar por más de una condición

Cuando queremos filtrar en base a dos condiciones podemos unirlos por los operadores:

“and” (&), que exige que se cumplan ambas condiciones para que filtra los casos:

```
base %>% filter(condicion1 & condicion2)
```

...o el operador “or” (|), que filtra los casos que cumple al menos una de las dos condiciones:

```
base %>% filter(condicion1 | condicion2)
```

A la función `filter()` es posible agregarle las condiciones “and” separadas por coma o en sucesivos filtrados:

```
base %>% filter(condicion1 & condicion2)
```

...es idéntico a ...

```
base %>% filter(condicion1, condicion2)
```

... y también a...

```
base %>% filter(condicion1) %>% filter(condicion2)
```

Mutate y case_when

Dentro de las posibilidades para trabajar dentro de la función `mutate()`, en clase se ha profundizado sobre `case_when()`. Función muy central a la hora de recodificar las categorías de nuestras variables, sea por una cuestión de

comodidad de lectura -de código número a texto- o para modificaciones como, por ejemplo, generar rangos de edad desde la variable edad.

```
base_covid <- base_covid %>%  
  select(edad) %>%  
  mutate(edad_rango = case_when(edad %in% c(0:18) ~ "0 a 18",  
                                edad %in% c(19:39) ~ "19 a 39",  
                                edad %in% c(40:59) ~ "40 a 59",  
                                edad >= 60 ~ "60 o más"))
```

Nota: el símbolo “~” se llama virgulilla, se puede realizar con *alt gr + 4* o *alt + 126*.

En una situación similar al ejemplo de los grupos de edad, surgió la consulta sobre una línea más al interior del `case_when()` con el fin de precisar un nuevo valor para casos que no estaban incluidos en la condiciones previamente establecidas.

Se explicaron dos resoluciones posibles para el caso.

- 1) Si existe una categoría residual con un valor específico -por ejemplo “SIN ESPECIFICAR” o valores negativos- se podría tomar ese valor e incluirlo dentro del `case_when()` para su recodificación:

```
mutate(edad_rango = case_when(edad %in% c(0:18) ~ "0 a 18",  
                              edad %in% c(19:39) ~ "19 a 39",  
                              edad %in% c(40:59) ~ "40 a 59",  
                              edad >= 60 ~ "60 o más",  
                              edad < 0 ~ "Ns/Nr"))
```

- 2) Otra opción es optar por el parámetro `TRUE` dentro del `case_when()`, el cual indica que para TODOS los casos -u observaciones- que NO cumplan ninguna de las condiciones previas se le asignará determinado valor, la línea sería así `TRUE ~ "Ns/Nr")` y se coloca como última condición dentro del `case_when()`.

Tener presente que esta función va “acomodando” cada una de las observaciones según la condición que cumplan en orden de cómo está armado el código, es decir que si no cumplen la primer condición, tratará con la segunda, y así sucesivamente hasta llegar al `TRUE`, en caso de estar presente. Para la opción 1 de este ejemplo solamente estamos agregando una condición extra con la que se incluirían -según nuestra base- todos los casos que no entran en el resto de las condiciones incluidas en el `case_when()`.

Uso de `group_by()` fuera de `summarise()`

Como hemos visto en clase, el `group_by()` es especialmente útil antes de realizar un `summarise()`. Sin embargo, este comando agrupa incluso dentro

de las otras funciones. Esto debemos recordarlo para no generar efectos no deseados en nuestros script.

Una función que puede aparecer en conjunto con `group_by` (aunque menos común que `summarise`) es `mutate()`. Para entender cuándo podríamos usarla pensemos el siguiente ejemplo. Tenemos una base con datos sobre personas que pertenecen a tres provincias y queremos agregar una nueva variable que diga, para cada persona, cuántas personas viven en dicha provincia. Una forma de hacer esto sería:

```
base <- base %>%  
  group_by(provincia) %>%  
  mutate(N = n())
```

A su vez, menos habitual, pero también posible, es que queramos usar nuestros datos agrupados mediante otras funciones. Supongamos que en nuestra base del ejemplo anterior tenemos también la edad de las personas y queremos retener sólo los casos de personas que tienen una edad igual o mayor que la media de edad de la provincia. Para ello podríamos hacer:

```
base <- base %>%  
  group_by(provincia) %>%  
  filter(edad >= mean(edad, na.rm = T) )
```

Aunque lo anterior podría hacerse en dos pasos (lo cual simplifica su lectura):

```
base <- base %>%  
  group_by(provincia) %>%  
  mutate(var_aux1 = mean(edad, na.rm = T) ) %>%  
  filter( edad >= var_aux1)
```

Funciones similares a `pivot_*`

Anteriormente en `tidyverse` (en el paquete `tidyr`) existían otras funciones que realizaban tareas similares a `pivot_longer()` y `pivot_wider()`:

`gather` similar a `pivot_longer`

`spread` similar a `pivot_wider`

También existían funciones similares en el paquete `reshape2`

Codificar variables mediante `*_join()`

Si bien en clase hemos usado para “codificar” nuestras regiones de la argentina vectores y `case_when()`, es posible realizar esta operación mediante un segundo data frame en el cual tengamos nuestros códigos relacionados con nuestras provincias. Para ello podríamos usar las funciones `*_join()` de `tidyverse` (`left_join`, `inner_join`, `right_join`, `full_join`), o bien la función `merge()` (dentro del paquete `base` de R). Sólo a modo de ejemplo, si

tuviéramos un data frame con dos columnas (llamado por ejemplo “codigos”), en donde tenemos los nombres de las provincias (variable que queremos codificar) y la región (código que queremos incorporar), podríamos utilizar el siguiente código para codificar nuestras provincias:

```
base_covid <- base_covid %>%  
  left_join(codigos, by = "provincia")
```

Este código nos dará como resultado un nuevo data frame, donde se incorporará para cada columna del data frame original (“base_covid”) que esté presente en la columna provincia del segundo (“codigo”) un nuevo valor en la variable region.

Como podrán prever, en realidad esta función no está diseñada específicamente para codificar, sino para unir o “mergear” bases en función de alguna variable común. Por ello, su funcionalidad puede ser mucho más amplia. No dude en usar la ayuda (por ejemplo `?left_join`) si les interesa conocer más sobre esta función.

3. Misceláneas: De todo un poco

Sobre el uso del foro

Si es posible, intenten hacer preguntas y proponer dudas en el foro durante la semana, porque es una manera de que todos puedan ver luego las respuestas. Recordemos que siempre hay muchas maneras de resolver un problema y ver cómo lo resuelve otro puede ayudarnos. Cada uno de nosotros irá encontrando la que le sea más cómoda. No hay soluciones mejores o peores, sino soluciones que resuelven nuestro problema.

¿Existe algo así como `base_covid(head)` que me tire sólo las 5 primeras filas?

Sí, pero recordemos que en R el orden es al revés al propuesto, es decir primero la función y luego, como uno de los argumentos, la base a la cual queremos aplicar dicha función. En este caso el código podría ser:

```
head(base, n = 5)
```

O bien usando el pipe:

```
library(tidyverse)  
base_covid %>% head(5)
```

¿Cómo incluir los valores NA en una tabla?

Ambos códigos permiten incluir los valores NA cuando usamos `table()`:

```
table(base$variable, exclude = NULL)  
table(base$variable, useNA = "ifany")
```

¿Cómo hacer el atajo de teclado para el pipe?

Ctrl + Shift + m

Con *Alt + Shift + k* se pueden ver todos los atajos de teclado.

Respecto a cómo funciona el pipe

Matemáticamente el pipe se comporta como una composición de funciones, de manera que si tenemos $f(g(x))$ con pipe el código nos queda:

```
x %>% g() %>% f()
```


¿Cómo levanto el environment que trabaje en un proyecto? Cuando abro el proyecto me viene vacío.

Para poder levantar un ambiente trabajado anteriormente es necesario tenerlo guardado en un archivo `.RData`. En realidad estos archivos solamente guarda los objetos, pero no las librerías que estaban cargadas y otros elementos que también son parte del ambiente. Estos archivos con los objetos pueden guardarse con el siguiente comando:

```
save.image("archivo.RData")
```

A su vez, para recuperar los objetos guardados puede ejecutarse:

```
load("archivo.RData")
```

De todas formas, es interesante señalar que en general suele ser recomendable no guardar nuestros objetos (resultados), sino guardar el “proceso” mediante el cual obtenemos estos resultados. Esto nos asegura la reproductibilidad de nuestro trabajo y es una de las razones por las que es tan importante tener un buen script. Siguiendo este consejo la mejor manera de recuperar nuestros objetos trabajados anteriormente es ejecutar nuevamente el script. Existen, sin embargo, algunas excepciones a esta sugerencia de no guardar objetos en los archivos `RData`, sobre todo cuando los procesos son muy “pesados” informáticamente y/o puede convenir realizarlos fuera del script.

¿Es posible ejecutar un script desde otro script?

Para esto podemos usar la función `source()` que nos permite ejecutar un archivo `R` de nuestro disco. Esto puede ser útil si queremos estructurar o dividir nuestro análisis en múltiples etapas y generar un script para cada una de ellas. Luego podríamos tener un “meta script” que ejecute cada uno de los script individuales.

Un ejemplo de esto sería llamar a un script para levantar datos (“levantar_datos.R”) desde nuestro script principal:

```
source("scripts/levantar_datos.R", encoding = "UTF-8")
```

Importación de bases con datos geoespaciales. ¿Cómo se abre un SHP file?

Estos temas los veremos en el próximo curso. Como adelanto contamos que en `R` existen librerías específicas para importar y trabajar datos espaciales (como los archivos *shape*). Una de las librerías más difundidas para trabajar con vectores es `sf`, la cual tiene la ventaja de que mantiene una estructura de trabajo similar a la que se utiliza en los data frame. Esto último es muy útil porque permite usar muchas de las operaciones de transformación y

graficación de datos de `tidyverse` (en particular permite graficar datos espaciales con `ggplot!`).

Más allá de este paquete, es importante señalar que en R existen diferentes librerías, cada una de las cuales tiene una forma diferente de ordenar los objetos geográficos.

Si les interesa profundizar o adelantar sobre estos temas pueden mirar la página de `r-spatial` donde se aborda el paquete `sf` (<https://r-spatial.github.io/sf/>). Además el libro *Geocomputation with R* (<https://geocompr.robinlovelace.net/>) es un aporte interesante que también se enfoca en el paquete `sf` para el manejo de vectores.

Paquete lubridate

En los ejercicios de una de las prácticas se planteó la consigna de tomar la variable `fecha_fallecimiento` y “desarmarla” en dos: año y mes. Una consecuencia de esta acción es perder la variable con formato “date” (fecha) y pasar a tener dos de otro tipo -numeric o character-. Se recomendó entonces trabajar con el paquete `lubridate` (<https://lubridate.tidyverse.org/>), que permite trabajar con variables de este tipo obviando dicho inconveniente y, además, incluye diversas funciones para trabajar con fechas y horas de manera más intuitiva y sencilla. Adjuntamos un ejemplo:

```
library(lubridate)
ejemplo_mutate_fecha <- base_covid %>%
  select(fallecido, fecha_fallecimiento) %>%
  filter(fallecido == "SI") %>%
  mutate(anio = year(fecha_fallecimiento),
         mes = month(x = fecha_fallecimiento))
```