

# **Redacción de artículos académicos con R**

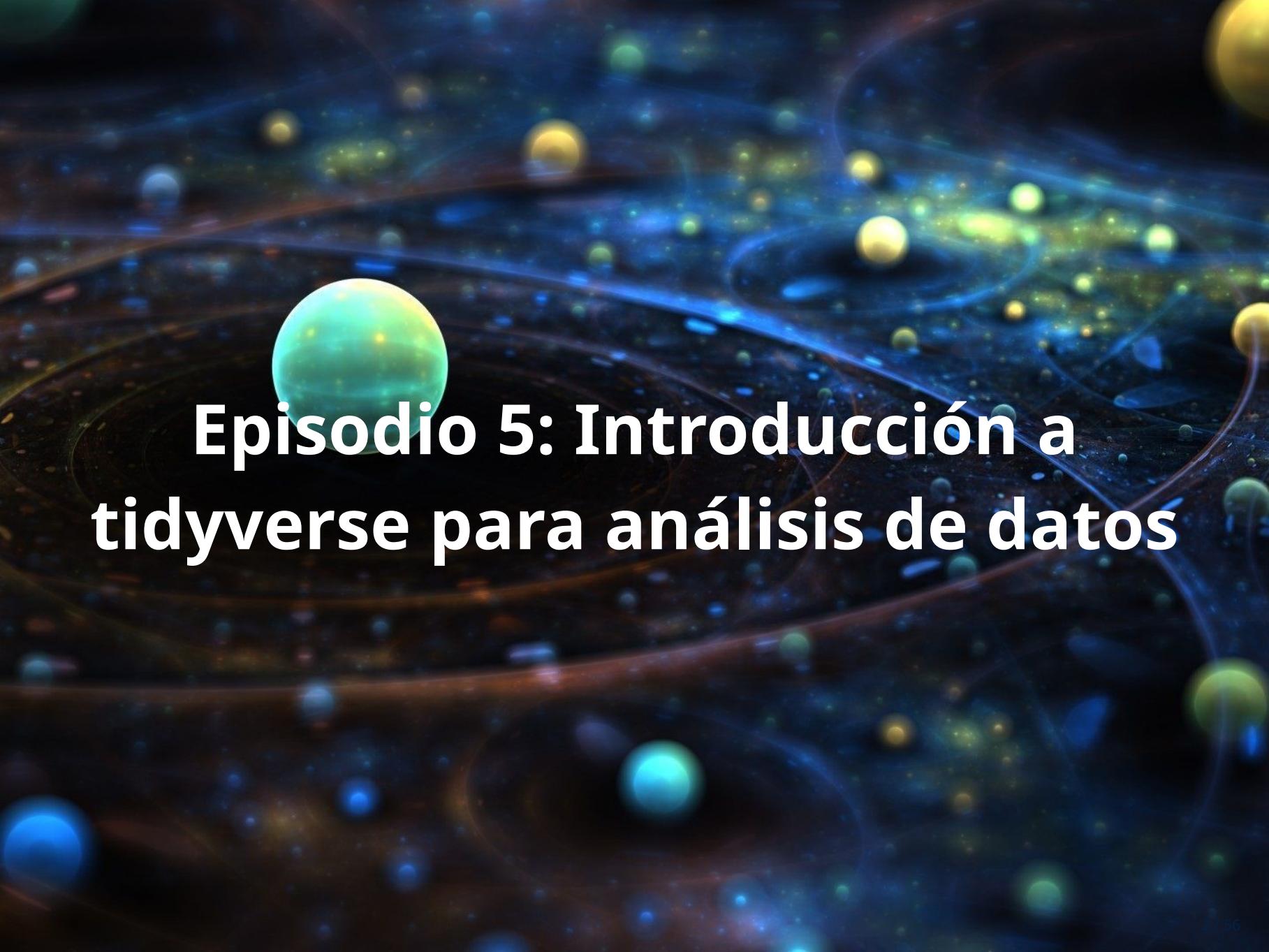
## **Episodio 5: Introducción a tidyverse para análisis de datos**

**Bajaña Alex**

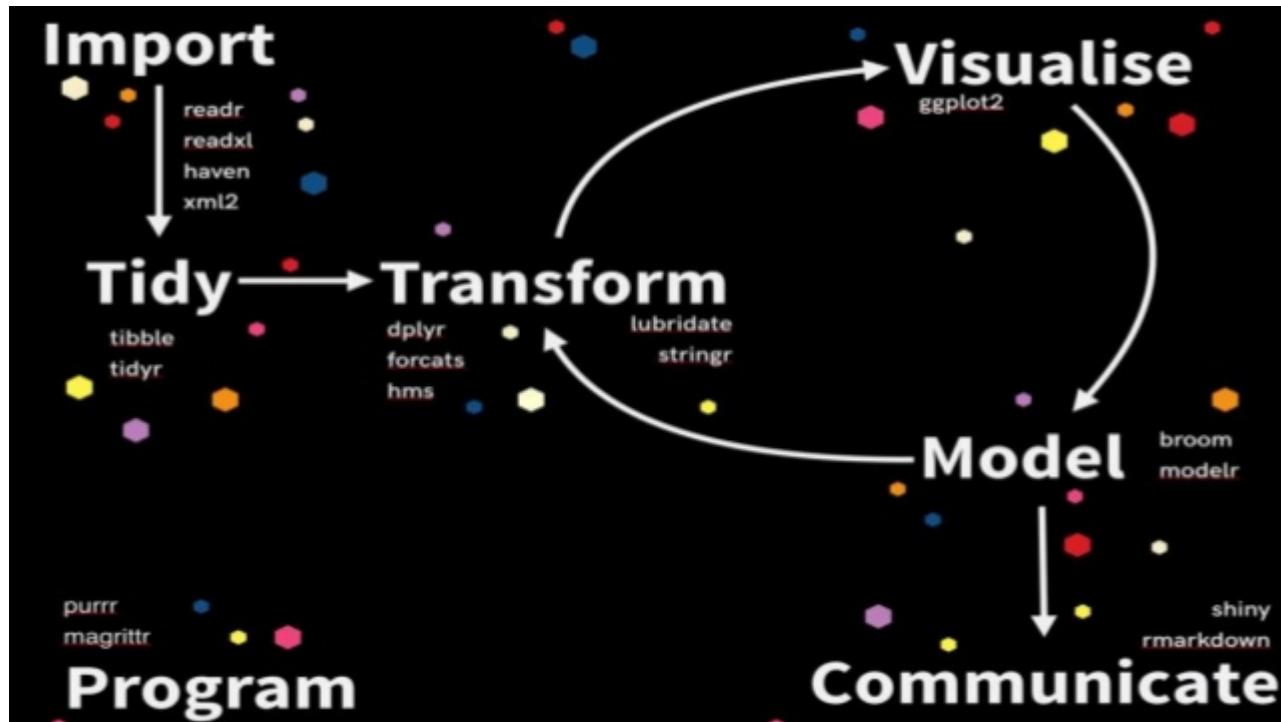
**Chanatasig Evelyn**

**Heredia Aracely**

**2022-07-09**



# Episodio 5: Introducción a tidyverse para análisis de datos



# El manifiesto tidy tools

# El manifiesto tidy tools

Es un documento que establece los principios del uso del paquete `tidyverse`. En base a ello, a continuación se muestran los cuatro principios:

- Reutilizar las estructuras de datos existentes.
- De funciones simples a complejas con la `pipe`, `%>%`
- Adoptar la programación funcional.
- Diseñar para humanos.

# Reutilizar las estructuras de datos existentes

Debido a que existen funciones desarrolladas previamente y comprobada su eficiencia y eficacia, es recomendable utilizarlas en lugar de crear nuevas.

Para hacer un correcto uso de las funciones ya existentes es necesario notar qué clase de objeto se debe declarar en cada argumento para que la función se ejecute correctamente.

Así también, se debe tener en cuenta las variables, objetos, listas, data frames, entre otros, creadas previamente en el `environment` para que puedan ser usadas a lo largo del `script`.

# De funciones simples a complejas con la pipe, %>%

Debido a que el usuario normalmente programa pedazos de código bien hecho pero, no están eficiente, para resolver este problema se tiene la `pipe` o su símbolo `%>%`, este sirve para unir estos pequeños pedazos de código.

Algunas cosas que tomaron en cuenta aquello que escribieron funciones son:

- Tratar de mantener las funciones lo más simples posible
- Los nombres de las funciones normalmente son verbos: `select()`, `name()`, etc.

# Adoptar la programación funcional

R es un lenguaje de programación funcional, no luches contra él. Esto significa principalmente que es indispensable **conocer el tipo de argumento** que necesita cada función, si se declara un **double** en una función cuyo argumento necesita **character**, la función no se va a ejecutar o no dará los resultados deseados.

Así también, existen otros lenguajes de programación como **Python** o **C** que tienen un lenguaje similar pero con ciertas diferencias, es por ello que se aconseja no pelear con el programa R sino, buscar la solución mediante la correcta declaración de los argumentos o buscando ayuda usando los métodos antes aprendidos.

# Diseñar para humanos

Diseñe su función principalmente para que sea fácil de usar y entender por otros usuarios. A pesar de que la persona entienda fácilmente lo que está programando, es posible que cuando otro usuario quiera leer el mismo código no logre descifrarlo fácilmente y le tome mucho más tiempo entenderlo que programar de nuevo todo; es por ello que es recomendable ser lo más **detallado y explicativo** posible por cada pedazo de código, normalmente se explica mediante comentarios de la acción que realiza cada parte. Esto se conoce como **reproducibilidad**.

La eficiencia de la computadora pasa a ser una preocupación secundaria porque en la mayoría de los análisis de datos, lo complejo es el tiempo de pensar en lo que se quiere que haga el código o describir lo que hace, más no el tiempo de cálculo.

# Diseñar para humanos

También hay que tener en cuenta la **responsabilidad con los datos**, es decir, muchas veces los datos que se analizan son de carácter privado y difundir, tanto los datos en sí mismos como los resultados puede ocasionar daños a terceros, es por ello que, las entidades se respaldan de forma legal para asegurarse de que la información proporcionada no sea divulgada ni siquiera de forma accidental.

Así mismo hay que tener en cuenta que con el **análisis de datos se puede cambiar realidades** pues, en base a los resultados del análisis, las empresas o aquellos interesados toman decisiones.

En los casos en los que se haga funciones, es importante usar nombres extensos y explícitos en lugar de los nombres cortos e implícitos. Así mismo, cuando se hacen familias de funciones, estas se deben identificar con un prefijo común en lugar de con un sufijo común, ej: `geom_bar()`, `geom_point()`, etc.

# Librería

# Cargando la librería necesaria

```
library(tidyverse)

## — Attaching packages ——————  
## ✓ ggplot2 3.3.6      ✓ purrr   0.3.4  
## ✓ tibble  3.1.7      ✓ dplyr   1.0.9  
## ✓ tidyverse 1.2.0     ✓ stringr 1.4.0  
## ✓ readr   2.1.2      ✓ forcats 0.5.1

## — Conflicts ——————  
## ✗ readr::edition_get()    masks testthat::edition_get()  
## ✗ dplyr::filter()        masks stats::filter()  
## ✗ purrr::is_null()       masks testthat::is_null()  
## ✗ dplyr::lag()           masks stats::lag()  
## ✗ readr::local_edition() masks testthat::local_edition()  
## ✗ dplyr::matches()       masks tidyverse::matches(), testthat::matches()
```

# Cargar datos

# Cargamos la base de datos:

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

## 'data.frame': 59208 obs. of 6 variables:
## $ sexo      : Factor w/ 2 levels "Hombre","Mujer": 1 2 1 1 2 2 2 1 1 2 ...
## $ edad      : int 34 23 6 28 26 3 39 22 10 4 ...
## $ a_trabajo : int 15 3 NA 5 NA NA NA NA NA ...
## $ nivel_inst: Factor w/ 9 levels "Ninguno","Centro de alfabetización",...: 5 4 4 5 3 NA 5 6 4 ...
## $ h_trabajo : int 45 40 NA 50 NA NA NA NA NA ...
## $ ingresos  : int 500 394 NA NA NA NA NA NA NA ...
```

```
#Renombramos la base
tabla <- enemdu

#Borramos lo que no necesitamos
rm("diccionario","enemdu","niv_inst","sexo")
```



# Pipe

# Pipe

Hasta ahora, cada vez que queríamos aplicar más de una función, se la declaraba como una secuencia de forma anidada, de la siguiente manera:

```
# Ejemplo de tres funciones anidadas a "x"  
tercera_función(segunda_función(primer_función(x)))
```

El operador pipe que se escribe de la siguiente manera: `%>%`; es útil para concatenar múltiples operaciones.

## Reproducibilidad:

La `pipe` permite "separar" el código en partes para que pueda ser comentado por secciones y aún así seguir funcionando. Esto es especialmente útil para la **reproducibilidad** pues permite a otros entender qué hace el código que hemos programado, así como también, nos permite a nosotros saber con mayor facilidad qué hace un código muy extenso. Para su mejor comprensión, se presentará un ejemplo en la siguiente clase.

# Sintaxis

La sintaxis the pipe, permite expresar de forma clara una secuencia de múltiples operaciones:

```
#Ejemplo de las 3 funciones aplicadas a "x" pero usando la pipe  
primera_función(x) %>% segunda_función(x) %>% tercera_función(x)
```

```
#También se pueden dar saltos de línea después de la pipe  
primera_función(x) %>%  
  segunda_función(x) %>%  
  tercera_función(x)
```

El atajo de teclado para el operador `%>%` es:

 (Mac)

 (Windows)

# Tibbles



# data.frames

# Tibbles

El paquete `tibble` nos hace la vida más fácil puesto que, un `tibble` es una versión moderna de los `data.frame`. Nos brinda más atributos, lo que a su vez, provee una versión de `data.frame` que facilita el trabajo con el tidyverse.

Aquí se puede observar que el `data.frame` llamado `tabla` sólo posee el atributo `data.frame`

```
class(tabla)
```

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

El mismo objeto pasa a tener más atributos al convertirlo en `tibble`.

```
library(tibble)
tabla_tibble <- as_tibble(tabla)
class(tabla_tibble)
```

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

Para convertir un `data.frame` en una tibble se emplea el comando `as_tibble()`

# Tibbles vs. data.frame

Existen dos diferencias principales entre el uso de un `tibble` y un `data.frame` clásico: la impresión en la consola y la selección de los subconjuntos.

## Impresión en la consola

Los `tibble` tienen un práctico método de impresión en la consola puesto que solo muestran las primeras 10 filas y solo aquellas columnas que entran en el ancho de la pantalla. Esto simplifica y facilita trabajar con bases de datos grandes. Además del nombre, cada columna muestra su tipo.

## Selección de subconjuntos

Si se quiere extraer una variable individual mediante el nombre, se necesitan el signo de dólar (\$).

En comparación a un `data.frame`, los `tibbles` son más estrictos, nunca funcionan con coincidencias parciales y generan una advertencia si la columna a la que intentas de acceder no existe.



**Manipulando los datos**

# Dplyr

Paquete para transformación de datos en forma **tabular**, integra en R una gramática intuitiva para transformarlos.

## Reduciendo una base de datos:

- **rename()**: Renombrar
- **select()**: Extrae variables
- **filter()**: Extrae casos
- **arrange()**: Reordena

# Renombrando una variable

# rename()



# Renombrar una variable

La función `rename()` esta diseñada para renombrar una variable en un `data.frame` de una forma más fácil.

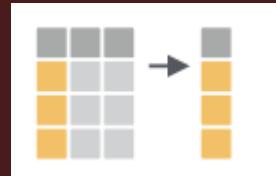
```
rename(tabla, Sexo = sexo,  
       Edad = edad,  
       AniosTrabajo = a_trabajo,  
       NivelInst = nivel_inst,  
       HorasTrab = h_trabajo,  
       Ingresos = ingresos) %>%  
head(4)
```

```
##      Sexo Edad AniosTrabajo      NivelInst HorasTrab Ingresos  
## 1 Hombre   34          15 Secundaria        45     500  
## 2 Mujer    23           3 Educación básica    40     394  
## 3 Hombre    6           NA Educación básica     NA      NA  
## 4 Hombre   28           5 Secundaria        50      NA
```



# Seleccionando los datos

# select()



# Selecciona o excluye variables de la base de datos

Con la función select podemos seleccionar o excluir columnas de un data frame:

```
select(.data, ...)
```

- `.data`: Data frame / tibble
- `...` : Nombres de columna, o función auxiliar

En todos los paquetes del `tidyverse` está integrada la `tidy evaluation`. Este es un método de evaluación que da una variedad de opciones para programar.

# Selecciona o excluye variables de la base de datos

```
# Nombre de variables
select(.data, nombre1, nombre2)

# Caracteres
select(.data, "nombre1", "nombre2")

# Posiciones
select(.data, c(1, 2))

# Todas menos ...
select(.data, -nombre1, -nombre2)
```

# Evaluación con nombres de variables:

```
library(dplyr)
select(tabla, h_trabajo, nivel_inst, ingresos) %>%
  head(7)
```

```
##   h_trabajo      nivel_inst  ingresos
## 1      45        Secundaria     500
## 2      40 Educación básica    394
## 3      NA Educación básica     NA
## 4      50        Secundaria     NA
## 5      NA         Primaria     NA
## 6      NA           <NA>       NA
## 7      NA        Secundaria     NA
```

# Funciones auxiliares

- **Rango de variables**

```
select(tabla, sexo:nivel_inst)
```

- **Todas menos**

```
select(tabla, -c(sexo,nivel_inst))
```

- **starts\_with():** Elegir aquellas que comienzan con "una expresión específica"

```
select(tabla, starts_with("a"))
```

- **ends\_with():** Elegir aquellas que terminan con "una expresión específica"

```
select(tabla, ends_with("o"))
```

# Funciones auxiliares

- **contains()**: Contiene una expresión específica

```
select(tabla, contains("os"))
```

- **matches()**: Usa una expresión regular, en este caso, un string de 4 caracteres

```
select(tabla, matches("^.{4}"))
```

- **one\_of()**: Elegir uno de

```
select(tabla, one_of(c("Edad", "edad", "EdaD")))
```

- **num\_range()**

```
tabla_2 <- tabla  
naedad(tabla_2) <- paste0("v_", 1:ncol(tabla_2))  
select(tabla_2, num_range("v_", 3:6))
```

# Prueba

Cuales de estas expresiones dan como resultado un set de datos con las variables: `sexo`, `a_trabajo`, `nivel_inst` y `ingresos`

```
select(tabla,-c(edad,h_trabajo))
```

```
select(tabla,sexo:nivel_inst)
```

```
select(tabla,matches("o"))
```

```
select(tabla, ends_with("o"))
```

```
select(tabla, c(1,3,4,6))
```

# Errores comunes

# Errores comunes:

```
# Como se trabaja con los nombres de las variables, es importante
# escribirlos exactamente como están en la base

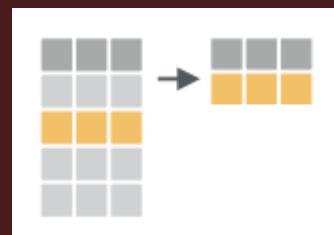
# En caso en que estén escritos de forma que no lee R, es importante
# usar los Nombres sintácticos
# (encerrar el nombre sintáctico entre ' ').

#Ejemplo:
select(tabla, `# hijos`)

# Incorrecto
select(tabla, # hijos)
```

# Filtrando los datos

# filter()



# Recordando los operadores lógicos:

Las condiciones pueden ser expresiones lógicas construidas mediante los operadores relacionales y lógicos:

Operador	Descripción
$x < y$	Menor que
$x > y$	Mayor que
$x == y$	Igual a
$x <= y$	Menor o igual
$x >= y$	Mayor o igual

Operador	Descripción
$x != y$	Todos excepto
$x %in% y$	Pertenece a
<code>is.na(x)</code>	Es NA?
<code>!is.na(x)</code>	No es NA?
$x %% y$	Módulo o residuo de la división
$x \%/\% y$	Parte entera de la división

# Filtrar la bases de datos

La función `filter()` nos permite filtrar filas según una o varias condiciones:

```
filter(.data, ... )
```

- `.data`: dataframe o tibble
- `... : operación lógica`

## Filtrar usando operadores lógicos. Ej:

```
#x == y: Igual a
filter(tabla, sexo == "Mujer") %>% head(3)
```

```
##   sexo edad a_trabajo      nivel_inst h_trabajo ingresos
## 1 Mujer  23       3 Educación básica        40     394
## 2 Mujer  26       NA          Primaria       NA       NA
## 3 Mujer   3       NA           <NA>       NA       NA
```

# Filtrar usando operadores lógicos. Ej:

```
# x >= y: Mayor o igual  
filter(tabla, h_trabajo >= 50) %>% head(2)  
  
##      sexo edad a_trabajo nivel_inst h_trabajo ingresos  
## 1 Hombre   28      5 Secundaria      50       NA  
## 2 Hombre   57     15 Primaria      50       NA  
  
# is.na(x): Es NA?  
ind <- sample(1:nrow(tabla), 30)  
tabla$edad[ind] <- NA_real_  
filter(tabla, is.na(edad)) %>% head(2)  
  
##      sexo edad a_trabajo      nivel_inst h_trabajo ingresos  
## 1 Mujer   NA      4      Primaria      20       NA  
## 2 Mujer   NA     NA Educación básica      NA       NA
```

# Filtrar usando operadores lógicos. Ej:

```
ind <- grepl(x = tabla$nivel_inst, pattern = "^\$")
f_tabla <- filter(tabla, ind)
table(f_tabla$nivel_inst)
```

```
# x == y: Igual a
filter(tabla, sexo == "Hombre", edad == 23) %>% head(4)
```

```
##      sexo edad a_trabajo      nivel_inst h_trabajo ingresos
## 1 Hombre  23       NA Superior universitaria     NA      NA
## 2 Hombre  23        5 Superior universitaria     20      NA
## 3 Hombre  23        0 Superior universitaria      5      40
## 4 Hombre  23        0 Superior universitaria     40    550
```

# Filtrar usando dos argumentos.

```
# Comando largo y menos eficiente
filter(tabla, sexo == "Hombre",
       edad == 18 | edad == 50,
       h_trabajo == 45,
       ingresos >= 600)
```

```
## sexo edad a_trabajo nivel_inst h_trabajo ingresos
## 1 Hombre 50 20 Secundaria 45 650
## 2 Hombre 50 30 Secundaria 45 800
```

```
# Comando eficiente
filter(tabla, sexo == "Hombre",
       edad %in% c(18, 50),
       h_trabajo == 45,
       ingresos >= 600)
```

# Errores comunes

# Errores comunes:

```
# No olvidar usar doble igual para equivalencia:  
filter(tabla, sexo = "Hombre")  
filter(tabla, sexo == "Hombre")  
  
# Los strings siempre van con comillas  
filter(tabla, sexo == Hombre)  
filter(tabla, sexo == "Hombre")  
  
# No se pueden hacer dos operadores en uno solo  
filter(tabla, 200 <ingresos < 500)  
filter(tabla, 200 <ingresos,ingresos < 500)  
# o  
filter(tabla, between(ingresos,200, 500))  
  
# Muchas equivalencias, entonces usar %in%  
filter(tabla,edad == 18 |edad == 25 |edad == 35 |edad == 50)  
filter(tabla,edad %in% c(18, 25, 35, 50))
```



# Recordando las operaciones algebráicas

Operador	Se lee	Descripción
$a \ \& \ b$	and	Intersección
$a \mid b$	or	Unión
$\text{xor}(a,b)$	exactly or	Diferencia
$!a$	not	Negación

Estas operaciones nos pueden ayudar con la selección de individuos o grupos de ellos dentro de nuestro conjunto de datos.

# Ejemplo con operaciones algebráicas

- Intersección de condiciones.

```
filter(nivel_inst == "Primaria", sexo == "Mujer")
```

- Primera condición verdadera, segunda falsa.

```
filter(nivel_inst == "Primaria", !sexo == "Mujer")
```

- Unión

```
filter(nivel_inst == "Primaria" | sexo == "Mujer")
```

- Solo una de las dos condiciones se cumple:

```
filter(xor(nivel_inst == "Primaria", sexo == "Mujer"))
```

## Prueba 2:

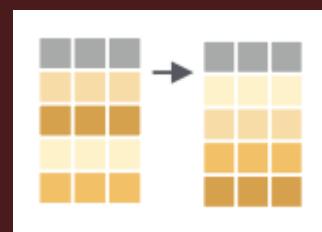
¿Cuál es el rango de ingresos de los hombres con un nivel de educación Secundaria en edades de 18 a 65 años?

**Hint:** sobre un vector numérico la función `min` devuelve el mínimo y `máx` el máximo



# Ordenando los datos

# arrange()



# Ordenar la base de datos

La función `arrange()` se utiliza para ordenar las filas de un data frame de acuerdo a una o varias columnas/variables.

```
arrange(.data, ...)
```

- `.data`: Data frame / tibble
- `...` : Nombres de columna, o función auxiliar

Por defecto `arrange()` ordena las filas por orden ascendente:

# Por default

```
# Orden ascendente:  
arrange(tabla, h_trabajo) %>% select(sexo,h_trabajo) %>% head(2)
```

```
##     sexo h_trabajo  
## 1 Mujer      1  
## 2 Hombre      1
```

# Ordenar la base de datos de forma descendente

```
# Orden descendente:  
arrange(tabla,desc(h_trabajo)) %>% select(sexo,h_trabajo) %>% head(2)
```

```
##     sexo h_trabajo  
## 1 Hombre    119  
## 2 Hombre    108
```

# Ordenar la base usando dos variables

Podemos ordenar las filas según varias variables, sin embargo hay que tener en cuenta que el orden con que se declaren las variables es importante debido a que influye en el resultado final. Por ejemplo:

```
arrange(tabla,h_trabajo,edad) %>% head(3)
```

	sexo	edad	a_trabajo	nivel_inst	h_trabajo	ingresos
1	Hombre	6		1 Educación básica	1	NA
2	Mujer	12		2 Educación básica	1	NA
3	Hombre	14		2 Educación básica	1	NA

```
arrange(tabla,edad,h_trabajo) %>% head(3)
```

	sexo	edad	a_trabajo	nivel_inst	h_trabajo	ingresos
1	Hombre	0		NA	<NA>	NA
2	Hombre	0		NA	<NA>	NA
3	Hombre	0		NA	<NA>	NA



# Recursos

- Programación en R
- Tibbles vs. data.frame
- El manifiesto tidy tools