

Dataframes

Jorge Loría

Creando el primero

Muchas veces se quieren ordenar datos de forma rectangular (como una matriz), pero se quiere que sean de varios tipos. Por ejemplo:

Creando el primero

Muchas veces se quieren ordenar datos de forma rectangular (como una matriz), pero se quiere que sean de varios tipos. Por ejemplo:

```
personas <- data.frame(Nombre = c('Pablo', 'Ana',  
                                   'Fernando', 'Maria'),  
                       Edad = c(26, 25, 19, 24),  
                       Sexo = c('M', 'F', 'M', 'F'),  
                       Provincia = c('San Jose', 'Cartago',  
                                     'Limon', 'Limon'),  
                       Asegurado = c(TRUE, TRUE, FALSE, FALSE))  
personas
```

##		Nombre	Edad	Sexo	Provincia	Asegurado
## 1		Pablo	26	M	San Jose	TRUE
## 2		Ana	25	F	Cartago	TRUE
## 3		Fernando	19	M	Limon	FALSE
## 4		Maria	24	F	Limon	FALSE

En muchos sentidos se comportan como una lista, pero con la restricción de que cada una de sus entradas debe ser un vector con la misma cantidad de entradas.

En muchos sentidos se comportan como una lista, pero con la restricción de que cada una de sus entradas debe ser un vector con la misma cantidad de entradas. Por ejemplo, para acceder una variable se puede utilizar \$:

```
personas$Nombre
```

```
## [1] Pablo      Ana          Fernando Maria  
## Levels: Ana Fernando Maria Pablo
```

Pero también se comportan como matriz:

```
personas[1:2,1:3]
```

```
##      Nombre Edad Sexo  
## 1  Pablo    26    M  
## 2   Ana     25    F
```

Y si se quieren varias columnas, completas, se pueden acceder de esta forma:

```
personas[3:4]
```

##		Sexo	Provincia
##	1	M	San Jose
##	2	F	Cartago
##	3	M	Limon
##	4	F	Limon

Y si se quieren varias columnas, completas, se pueden acceder de esta forma:

```
personas[3:4]
```

```
##      Sexo Provincia
## 1      M   San Jose
## 2      F   Cartago
## 3      M     Limon
## 4      F     Limon
```

O se puede acceder a una columna como si el data.frame fuera, efectivamente, una lista:

```
personas[[2]]
```

```
## [1] 26 25 19 24
```

Se accesa con vectores

Por lo que se puede acceder con un vector así:

```
personas[c(1,4,5)]
```

##		Nombre	Provincia	Asegurado
## 1	Pablo	San Jose	TRUE	
## 2	Ana	Cartago	TRUE	
## 3	Fernando	Limon	FALSE	
## 4	Maria	Limon	FALSE	

También funciona como matriz/vector

Si se quieren acceder ciertas entradas de un vector, se puede hacer de la siguiente forma:

```
personas[personas$Sexo == 'F',]
```

```
##      Nombre Edad Sexo Provincia Asegurado
## 2      Ana   25    F    Cartago      TRUE
## 4    Maria   24    F     Limon     FALSE
```

Note la coma, si esta no se coloca se obtiene un resultado inesperado.

También funciona como matriz/vector

Si se quieren acceder ciertas entradas de un vector, se puede hacer de la siguiente forma:

```
personas[personas$Sexo == 'F',]
```

```
##      Nombre Edad Sexo Provincia Asegurado
## 2      Ana   25    F    Cartago      TRUE
## 4     Maria   24    F      Limon     FALSE
```

Note la coma, si esta no se coloca se obtiene un resultado inesperado. ¿Qué se obtiene si no se pone la coma? Obtenga las personas cuya edad es mayor a 20.

También funciona como matriz/vector

Si se quieren acceder ciertas entradas de un vector, se puede hacer de la siguiente forma:

```
personas[personas$Sexo == 'F',]
```

```
##      Nombre Edad Sexo Provincia Asegurado
## 2      Ana   25    F   Cartago      TRUE
## 4     Maria  24    F    Limon     FALSE
```

Note la coma, si esta no se coloca se obtiene un resultado inesperado. ¿Qué se obtiene si no se pone la coma? Obtenga las personas cuya edad es mayor a 20.

```
personas[personas$Edad>20,]
```

```
##      Nombre Edad Sexo Provincia Asegurado
## 1     Pablo   26    M   San Jose      TRUE
## 2      Ana   25    F   Cartago      TRUE
## 4     Maria  24    F    Limon     FALSE
```

La forma en que R indexa es *inteligente*, por lo que si se quiere obtener únicamente los valores de una columna usando el método anterior se puede hacer:

```
personas[personas$Sexo == 'F', 'Edad']
```

```
## [1] 25 24
```

El cual da un vector como resultado.

La forma en que R indexa es *inteligente*, por lo que si se quiere obtener únicamente los valores de una columna usando el método anterior se puede hacer:

```
personas[personas$Sexo == 'F', 'Edad']
```

```
## [1] 25 24
```

El cual da un vector como resultado.

Obtenga el vector que corresponde a las provincias de las personas que están aseguradas.

La forma en que R indexa es *inteligente*, por lo que si se quiere obtener únicamente los valores de una columna usando el método anterior se puede hacer:

```
personas[personas$Sexo == 'F', 'Edad']
```

```
## [1] 25 24
```

El cual da un vector como resultado.

Obtenga el vector que corresponde a las provincias de las personas que están aseguradas.

```
personas[personas$Asegurad, 'Provincia']
```

```
## [1] San Jose Cartago
```

```
## Levels: Cartago Limon San Jose
```

Suave un toque

Si le pedimos el tipo (`typeof`) a la columna de Provincia, ¿qué obtenemos?

Suave un toque

Si le pedimos el tipo (typeof) a la columna de Provincia, ¿qué obtenemos?

```
typeof(personas$Provincia)
```

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

```
# typeof(personas[, 'Provincia'])  
# typeof(personas[, 4])  
# typeof(personas[[4]])
```

Pero... lo habíamos definido como un string, ¿por qué lo interpreta como un número?

Suave un toque

Si le pedimos el tipo (`typeof`) a la columna de Provincia, ¿qué obtenemos?

```
typeof(personas$Provincia)
```

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

```
# typeof(personas[, 'Provincia'])  
# typeof(personas[, 4])  
# typeof(personas[[4]])
```

Pero... lo habíamos definido como un string, ¿por qué lo interpreta como un número?

Resulta, que R en los dataframes se asume que los strings son *factors*.

Suave un toque

Si le pedimos el tipo (`typeof`) a la columna de Provincia, ¿qué obtenemos?

```
typeof(personas$Provincia)
```

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

```
# typeof(personas[, 'Provincia'])  
# typeof(personas[, 4])  
# typeof(personas[[4]])
```

Pero... lo habíamos definido como un string, ¿por qué lo interpreta como un número?

Resulta, que R en los dataframes se asume que los strings son *factors*. Ok, muy bien.

Pero, ¿qué es un factor?

Pausa de *factors*

Los *factors* son para variables (columnas) categóricas, o sea aquellas que indican que una variable tiene una cantidad discreta de opciones (por ejemplo, el sexo, la provincia).

```
x <- factor(c('San Jose', 'Alajuela', 'Cartago', 'San Jose'))  
x
```

```
## [1] San Jose Alajuela Cartago San Jose  
## Levels: Alajuela Cartago San Jose
```

Pausa de *factors*

Los *factors* son para variables (columnas) categóricas, o sea aquellas que indican que una variable tiene una cantidad discreta de opciones (por ejemplo, el sexo, la provincia).

```
x <- factor(c('San Jose', 'Alajuela', 'Cartago', 'San Jose'))  
x
```

```
## [1] San Jose Alajuela Cartago San Jose  
## Levels: Alajuela Cartago San Jose
```

También pueden forzarse valores numéricos a ser factores:

```
y <- factor(c(1,3,1,4,1,2,6,5))  
y
```

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

Con los enteros es **más** molesto

Pues los niveles que se asignan no necesariamente coinciden con los que realmente toman:

```
y
```

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

```
levels(y)
```

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

Si intentamos que lo interprete como numérico:

```
as.numeric(y)
```

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

Con los números es *aún más* molesto

```
z1 <- factor(c(0.7,1.9,15.6,1.4))  
z1
```

```
## [1] 0.7  1.9 15.6 1.4  
## Levels: 0.7 1.4 1.9 15.6
```

Con los números es *aún más* molesto

```
z1 <- factor(c(0.7,1.9,15.6,1.4))  
z1
```

```
## [1] 0.7  1.9  15.6 1.4  
## Levels: 0.7 1.4 1.9 15.6
```

```
levels(z1)
```

```
## [1] "0.7"  "1.4"  "1.9"  "15.6"
```

```
as.numeric(z1)
```

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

Con los números es *aún más* molesto

```
z1 <- factor(c(0.7,1.9,15.6,1.4))  
z1
```

```
## [1] 0.7  1.9  15.6 1.4  
## Levels: 0.7 1.4 1.9 15.6
```

```
levels(z1)
```

```
## [1] "0.7"  "1.4"  "1.9"  "15.6"
```

```
as.numeric(z1)
```

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

```
as.numeric(levels(z1))
```

```
## [1] 0.7  1.4  1.9 15.6
```


Solución:

Solución:

```
as.numeric(levels(z1))[z1]
```

```
## [1]  0.7  1.9 15.6  1.4
```

Solución:

```
as.numeric(levels(z1))[z1]
```

```
## [1]  0.7  1.9 15.6  1.4
```

¿Cómo harían para hacer el llamado anterior usando %>%?

Solución:

```
as.numeric(levels(z1))[z1]
```

```
## [1] 0.7 1.9 15.6 1.4
```

¿Cómo harían para hacer el llamado anterior usando %>%?

```
library(magrittr)
z1 %>%
  levels() %>%
  as.numeric() %>%
  `[`(z1)
```

```
## [1] 0.7 1.9 15.6 1.4
```

Solución:

```
as.numeric(levels(z1))[z1]
```

```
## [1] 0.7 1.9 15.6 1.4
```

¿Cómo harían para hacer el llamado anterior usando %>%?

```
library(magrittr)
z1 %>%
  levels() %>%
  as.numeric() %>%
  `[`(z1)
```

```
## [1] 0.7 1.9 15.6 1.4
```

También se puede hacer:

```
z1 %>% as.character() %>% as.numeric()
```

Pausa de *factors*

Noten que los factores no tienen cómo saber si faltan variables, por ejemplo con las provincias solo se incluyen las 3 que indicamos, pero si luego quisiéramos *juntarle* observaciones con variables con nuevas provincias vamos a tener problemas:

```
x2 <- c(x, 'Heredia')  
x2
```

```
## [1] "3"      "1"      "2"      "3"      "Heredia"
```

Pues toma el valor que le está asignando internamente para referirse a los valores que tenía para x.

Pausa de *factors*

Noten que los factores no tienen cómo saber si faltan variables, por ejemplo con las provincias solo se incluyen las 3 que indicamos, pero si luego quisiéramos *juntarle* observaciones con variables con nuevas provincias vamos a tener problemas:

```
x2 <- c(x, 'Heredia')  
x2
```

```
## [1] "3"      "1"      "2"      "3"      "Heredia"
```

Pues toma el valor que le está asignando internamente para referirse a los valores que tenía para x. Para remediar esto, se usa `as.character`

```
x3 <- c(as.character(x), 'Heredia')  
x3
```

```
## [1] "San Jose" "Alajuela" "Cartago"   "San Jose" "Heredia"
```

Pausa de *factors*

Noten que los factores no tienen cómo saber si faltan variables, por ejemplo con las provincias solo se incluyen las 3 que indicamos, pero si luego quisiéramos *juntarle* observaciones con variables con nuevas provincias vamos a tener problemas:

```
x2 <- c(x, 'Heredia')  
x2
```

```
## [1] "3"      "1"      "2"      "3"      "Heredia"
```

Pues toma el valor que le está asignando internamente para referirse a los valores que tenía para x. Para remediar esto, se usa `as.character`

```
x3 <- c(as.character(x), 'Heredia')  
x3
```

```
## [1] "San Jose" "Alajuela" "Cartago"  "San Jose" "Heredia"
```

Para conocer todos los valores que toma un vector de factors, se usa la función `levels`.

Continuamos con dataframes

De forma predefinida, los *characters* en los dataframes se interpretan como *factors*, lo cual puede ser problemático:

```
str(personas)
```

```
## 'data.frame':    4 obs. of  5 variables:
##  $ Nombre      : Factor w/ 4 levels "Ana","Fernando",...: 4 1 2 3
##  $ Edad        : num  26 25 19 24
##  $ Sexo        : Factor w/ 2 levels "F","M": 2 1 2 1
##  $ Provincia: Factor w/ 3 levels "Cartago","Limon",...: 3 1 2 2
##  $ Asegurado: logi  TRUE TRUE FALSE FALSE
```

Por lo que, se puede mandar como parámetro opcional que los strings no sean *factors*, cambiando el parámetro `stringsAsFactors = FALSE`, al definir el `data.frame`

Por lo que, se puede mandar como parámetro opcional que los strings no sean *factors*, cambiando el parámetro `stringsAsFactors = FALSE`, al definir el `data.frame`

```
personas <- data.frame(Nombre = c('Pablo', 'Ana', 'Fernando', 'Maria'),
                        Edad = c(26, 25, 19, 24),
                        Sexo = c('M', 'F', 'M', 'F'),
                        Provincia = c('San Jose', 'Cartago', 'Limon', 'Limon'),
                        Asegurado = c(TRUE, TRUE, FALSE, FALSE),
                        stringsAsFactors = FALSE)

str(personas)
```

```
## 'data.frame':    4 obs. of  5 variables:
## $ Nombre      : chr  "Pablo" "Ana" "Fernando" "Maria"
## $ Edad        : num  26 25 19 24
## $ Sexo        : chr  "M" "F" "M" "F"
## $ Provincia: chr  "San Jose" "Cartago" "Limon" "Limon"
## $ Asegurado: logi  TRUE TRUE FALSE FALSE
```

Accesar vectores

Por lo que si ahora accesamos una de estas variables

```
personas$Nombre
```

```
## [1] "Pablo"      "Ana"        "Fernando"   "Maria"
```

```
personas$Provincia
```

```
## [1] "San Jose" "Cartago"   "Limon"     "Limon"
```

Obtenemos los valores propiamente, en lugar de obtener los niveles que obteníamos antes.

Accesar vectores

Por lo que si ahora accesamos una de estas variables

```
personas$Nombre
```

```
## [1] "Pablo"      "Ana"        "Fernando"   "Maria"
```

```
personas$Provincia
```

```
## [1] "San Jose" "Cartago"   "Limon"     "Limon"
```

Obtenemos los valores propiamente, en lugar de obtener los niveles que obteníamos antes. Revise que efectivamente el tipo de estas columnas es ahora character, usando la variable

Definir nuevas variables

Para definir nuevas variables en un dataframe es como lo hacíamos en una lista:

```
personas$Antiguedad <- c(5,3,1,2)
personas['Casado'] <- c(TRUE,TRUE,TRUE,FALSE)
personas['Apellido'] <- c('Gonzalez','Solano','Vargas','Solis')
str(personas)
```

```
## 'data.frame':    4 obs. of  8 variables:
## $ Nombre      : chr  "Pablo" "Ana" "Fernando" "Maria"
## $ Edad        : num  26 25 19 24
## $ Sexo        : chr  "M" "F" "M" "F"
## $ Provincia   : chr  "San Jose" "Cartago" "Limon" "Limon"
## $ Asegurado   : logi  TRUE TRUE FALSE FALSE
## $ Antiguedad  : num  5 3 1 2
## $ Casado      : logi  TRUE TRUE TRUE FALSE
## $ Apellido    : chr  "Gonzalez" "Solano" "Vargas" "Solis"
```

Dataframes con dplyr

Parte de lo que vamos a aprender es a utilizar los **verbos** que vienen en la librería dplyr. Que tiene bastantes opciones para la manipulación más sencilla de los data frames.

```
install.packages('dplyr')
```

```
library(dplyr)
```

Los verbos

Hay 6 verbos básicos en dplyr que vamos a ver con detenimiento:

función	acción
<code>filter</code>	conserva filas que cumplan la condición
<code>arrange</code>	ordena las filas según el orden
<code>select</code>	conserva/elimina las columnas por su nombre
<code>mutate</code>	crea nuevas variables con funciones de variables existentes
<code>summarise</code>	resume los datos
<code>group_by*</code>	agrupa bajo ciertas clasificaciones

Los verbos

Todos los verbos funcionan de una forma similar, en el sentido de que:

Los verbos

Todos los verbos funcionan de una forma similar, en el sentido de que:

- ▶ el primer argumento es un dataframe,

Los verbos

Todos los verbos funcionan de una forma similar, en el sentido de que:

- ▶ el primer argumento es un dataframe,
- ▶ los argumentos siguientes describen *qué* se va a hacer con el dataframe, usando las variables existentes (sin comillas).

Los verbos

Todos los verbos funcionan de una forma similar, en el sentido de que:

- ▶ el primer argumento es un dataframe,
- ▶ los argumentos siguientes describen *qué* se va a hacer con el dataframe, usando las variables existentes (sin comillas).
- ▶ el resultado es un dataframe

Los verbos

Todos los verbos funcionan de una forma similar, en el sentido de que:

- ▶ el primer argumento es un dataframe,
- ▶ los argumentos siguientes describen *qué* se va a hacer con el dataframe, usando las variables existentes (sin comillas).
- ▶ el resultado es un dataframe

Los verbos

Todos los verbos funcionan de una forma similar, en el sentido de que:

- ▶ el primer argumento es un dataframe,
- ▶ los argumentos siguientes describen *qué* se va a hacer con el dataframe, usando las variables existentes (sin comillas).
- ▶ el resultado es un dataframe

Todo esto permite que se concatenen (%>%) operaciones sencillas para obtener resultados complejos.

filter

filter



Figure 1: Otro tipo de filtro

filter

Esta función es muy similar a lo que ya hicimos hace varias diapositivas, pues permite eliminar/conservar filas según un criterio (o más), pero utilizando menos palabras.

filter

Esta función es muy similar a lo que ya hicimos hace varias diapositivas, pues permite eliminar/conservar filas según un criterio (o más), pero utilizando menos palabras. Por ejemplo, si queremos obtener las filas que corresponden a las personas aseguradas:

```
filter(personas,Asegurado)
```

##		Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1		Pablo	26	M	San Jose	TRUE	5	TRUE	Gonzalez
## 2		Ana	25	F	Cartago	TRUE	3	TRUE	Solano

filter

Esta función es muy similar a lo que ya hicimos hace varias diapositivas, pues permite eliminar/conservar filas según un criterio (o más), pero utilizando menos palabras. Por ejemplo, si queremos obtener las filas que corresponden a las personas aseguradas:

```
filter(personas,Asegurado)
```

```
##      Nombre Edad Sexo Provincia Asegurado Antigüedad Casado Apellido
## 1   Pablo   26    M   San Jose      TRUE           5    TRUE Gonzalez
## 2    Ana   25    F   Cartago       TRUE           3    TRUE   Solano
```

Si se quiere guardar en un nuevo dataframe, se guarda como cualquier otro objeto

```
asegurados_df <- personas %>% filter(Asegurado)
```

filter

Esta función es muy similar a lo que ya hicimos hace varias diapositivas, pues permite eliminar/conservar filas según un criterio (o más), pero utilizando menos palabras. Por ejemplo, si queremos obtener las filas que corresponden a las personas aseguradas:

```
filter(personas,Asegurado)
```

##	Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1	Pablo	26	M	San Jose	TRUE	5	TRUE	Gonzalez
## 2	Ana	25	F	Cartago	TRUE	3	TRUE	Solano

Si se quiere guardar en un nuevo dataframe, se guarda como cualquier otro objeto

```
asegurados_df <- personas %>% filter(Asegurado)
```

Es equivalente usar cualquiera de las dos notaciones: con %>% o sin este.

filter

Si quisiéramos únicamente ver las personas con sexo femenino, hacemos:

```
personas %>% filter(Sexo == 'F')
```

##	Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1	Ana	25	F	Cartago	TRUE	3	TRUE	Solano
## 2	Maria	24	F	Limon	FALSE	2	FALSE	Solis

filter

Si quisiéramos únicamente ver las personas con sexo femenino, hacemos:

```
personas %>% filter(Sexo == 'F')
```

##	Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1	Ana	25	F	Cartago	TRUE	3	TRUE	Solano
## 2	Maria	24	F	Limon	FALSE	2	FALSE	Solis

Usando filter, encuentre las personas con edad mayor a 23.

filter

Si quisiéramos únicamente ver las personas con sexo femenino, hacemos:

```
personas %>% filter(Sexo == 'F')
```

##	Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1	Ana	25	F	Cartago	TRUE	3	TRUE	Solano
## 2	Maria	24	F	Limon	FALSE	2	FALSE	Solis

Usando `filter`, encuentre las personas con edad mayor a 23. Ahora, busque únicamente las personas que son de Limon. Tenga cuidado si usó tildes al definirlo.

filter

Si quisiéramos únicamente ver las personas con sexo femenino, hacemos:

```
personas %>% filter(Sexo == 'F')
```

##	Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1	Ana	25	F	Cartago	TRUE	3	TRUE	Solano
## 2	Maria	24	F	Limon	FALSE	2	FALSE	Solis

Usando filter, encuentre las personas con edad mayor a 23. Ahora, busque únicamente las personas que son de Limon. Tenga cuidado si usó tildes al definirlo.

```
personas %>% filter(Edad > 23)
personas %>% filter(Provincia == 'Limon')
```


Interacción del `filter`

También se puede considerar una interacción entre las variables que se usan para definir un `filter`. Por ejemplo, si queremos las personas que comenzaron a trabajar con al menos 20 años ($\text{Edad} - \text{Antigüedad} \geq 20$) y que además no estén casados. Hacemos:

Interacción del filter

También se puede considerar una interacción entre las variables que se usan para definir un filter. Por ejemplo, si queremos las personas que comenzaron a trabajar con al menos 20 años ($\text{Edad} - \text{Antigüedad} \geq 20$) y que además no estén casados. Hacemos:

```
personas %>% filter(Edad - Antigüedad >= 20, !Casado)
```

```
##      Nombre Edad Sexo Provincia Asegurado Antigüedad Casado Apellido
## 1  Maria    24    F      Limon      FALSE           2  FALSE    Solis
```

Interacción del filter

También se puede considerar una interacción entre las variables que se usan para definir un filter. Por ejemplo, si queremos las personas que comenzaron a trabajar con al menos 20 años ($\text{Edad} - \text{Antigüedad} \geq 20$) y que además no estén casados. Hacemos:

```
personas %>% filter(Edad - Antigüedad >= 20, !Casado)
```

```
##      Nombre Edad Sexo Provincia Asegurado Antigüedad Casado Apellido
## 1  Maria    24    F      Limon      FALSE           2  FALSE    Solis
```

También se pudo hacer:

```
personas %>% filter((Edad - Antigüedad >= 20) & !Casado)
```

Interacción del filter

También se puede considerar una interacción entre las variables que se usan para definir un filter. Por ejemplo, si queremos las personas que comenzaron a trabajar con al menos 20 años ($\text{Edad} - \text{Antigüedad} \geq 20$) y que además no estén casados. Hacemos:

```
personas %>% filter(Edad - Antigüedad >= 20, !Casado)
```

```
##   Nombre Edad Sexo Provincia Asegurado Antigüedad Casado Apellido
## 1  Maria   24    F      Limon      FALSE           2  FALSE   Solis
```

También se pudo hacer:

```
personas %>% filter((Edad - Antigüedad >= 20) & !Casado)
```

Repita el ejercicio anterior, pero ahora que las personas hayan comenzado a trabajar con al menos 20 años, **o** que estén casados.

arrange

arrange

Este verbo ordena según las columnas que se le indiquen:

```
personas %>% arrange(Edad)
```

##	Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1	Fernando	19	M	Limon	FALSE	1	TRUE	Vargas
## 2	Maria	24	F	Limon	FALSE	2	FALSE	Solis
## 3	Ana	25	F	Cartago	TRUE	3	TRUE	Solano
## 4	Pablo	26	M	San Jose	TRUE	5	TRUE	Gonzalez

Ordena palabras

También puede ordenar lexicográficamente:

```
personas %>% arrange(Provincia)
```

##		Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1		Ana	25	F	Cartago	TRUE	3	TRUE	Solano
## 2		Fernando	19	M	Limon	FALSE	1	TRUE	Vargas
## 3		Maria	24	F	Limon	FALSE	2	FALSE	Solis
## 4		Pablo	26	M	San Jose	TRUE	5	TRUE	Gonzalez

Ordena palabras

También puede ordenar lexicográficamente:

```
personas %>% arrange(Provincia)
```

##		Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1		Ana	25	F	Cartago	TRUE	3	TRUE	Solano
## 2		Fernando	19	M	Limon	FALSE	1	TRUE	Vargas
## 3		Maria	24	F	Limon	FALSE	2	FALSE	Solis
## 4		Pablo	26	M	San Jose	TRUE	5	TRUE	Gonzalez

Note que sí podemos ordenar por characters y que en general podemos comparar dos strings 'A' > 'B'

Ordena palabras

También puede ordenar lexicográficamente:

```
personas %>% arrange(Provincia)
```

##		Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1		Ana	25	F	Cartago	TRUE	3	TRUE	Solano
## 2		Fernando	19	M	Limon	FALSE	1	TRUE	Vargas
## 3		Maria	24	F	Limon	FALSE	2	FALSE	Solis
## 4		Pablo	26	M	San Jose	TRUE	5	TRUE	Gonzalez

Note que sí podemos ordenar por characters y que en general podemos comparar dos strings 'A' > 'B', pero no podemos intentar hacer restas o sumas entre strings: 'A' + 'B', tira un error.

Ordena palabras

También puede ordenar lexicográficamente:

```
personas %>% arrange(Provincia)
```

##		Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1		Ana	25	F	Cartago	TRUE	3	TRUE	Solano
## 2		Fernando	19	M	Limon	FALSE	1	TRUE	Vargas
## 3		Maria	24	F	Limon	FALSE	2	FALSE	Solis
## 4		Pablo	26	M	San Jose	TRUE	5	TRUE	Gonzalez

Note que sí podemos ordenar por characters y que en general podemos comparar dos strings 'A' > 'B', pero no podemos intentar hacer restas o sumas entre strings: 'A' + 'B', tira un error.

Ordene a las personas por apellidos.

arrange descendiente

Para que quede en orden descendiente, se llama la misma columna pero envuelta en `desc(...)`:

```
personas %>% arrange(Provincia, desc(Edad))
```

##	Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1	Ana	25	F	Cartago	TRUE	3	TRUE	Solano
## 2	Maria	24	F	Limon	FALSE	2	FALSE	Solis
## 3	Fernando	19	M	Limon	FALSE	1	TRUE	Vargas
## 4	Pablo	26	M	San Jose	TRUE	5	TRUE	Gonzalez

También se puede ordenar con más de una columna, al ponerla como otros parámetros de la función. Siempre se le da prioridad a la primera columna que se indica, los empates los resuelve la segunda, los empates de la segunda los resuelve la tercera y así sucesivamente...

arrange, más complejo

Se pueden hacer operaciones (un poco) más complejas con arrange:

arrange, más complejo

Se pueden hacer operaciones (un poco) más complejas con arrange:

```
personas %>% arrange(abs(Edad - Antigüedad - 20))
```

##	Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1	Pablo	26	M	San Jose	TRUE	5	TRUE	Gonzalez
## 2	Ana	25	F	Cartago	TRUE	3	TRUE	Solano
## 3	Fernando	19	M	Limon	FALSE	1	TRUE	Vargas
## 4	Maria	24	F	Limon	FALSE	2	FALSE	Solis

¿Qué indica el orden anterior?

arrange, más complejo

Se pueden hacer operaciones (un poco) más complejas con arrange:

```
personas %>% arrange(abs(Edad - Antigüedad - 20))
```

##	Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1	Pablo	26	M	San Jose	TRUE	5	TRUE	Gonzalez
## 2	Ana	25	F	Cartago	TRUE	3	TRUE	Solano
## 3	Fernando	19	M	Limon	FALSE	1	TRUE	Vargas
## 4	Maria	24	F	Limon	FALSE	2	FALSE	Solis

¿Qué indica el orden anterior? Lo anterior indica quienes son los que están más cercanos de haber comenzado a trabajar a los 20 años.

arrange, más complejo

Se pueden hacer operaciones (un poco) más complejas con arrange:

```
personas %>% arrange(abs(Edad - Antigüedad - 20))
```

##	Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1	Pablo	26	M	San Jose	TRUE	5	TRUE	Gonzalez
## 2	Ana	25	F	Cartago	TRUE	3	TRUE	Solano
## 3	Fernando	19	M	Limon	FALSE	1	TRUE	Vargas
## 4	Maria	24	F	Limon	FALSE	2	FALSE	Solis

¿Qué indica el orden anterior? Lo anterior indica quienes son los que están más cercanos de haber comenzado a trabajar a los 20 años.

Ordene a las personas primero por su estado de asegurado, y luego por antigüedad.

¿Qué va primero? ¿True o False?

arrange, más complejo

Se pueden hacer operaciones (un poco) más complejas con arrange:

```
personas %>% arrange(abs(Edad - Antigüedad - 20))
```

##	Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1	Pablo	26	M	San Jose	TRUE	5	TRUE	Gonzalez
## 2	Ana	25	F	Cartago	TRUE	3	TRUE	Solano
## 3	Fernando	19	M	Limon	FALSE	1	TRUE	Vargas
## 4	Maria	24	F	Limon	FALSE	2	FALSE	Solis

¿Qué indica el orden anterior? Lo anterior indica quienes son los que están más cercanos de haber comenzado a trabajar a los 20 años.

Ordene a las personas primero por su estado de asegurado, y luego por antigüedad. ¿Qué va primero? ¿True o False? Si se quisiera ordenar por sexo, ¿cuál va primero? Realice este orden, y luego por nombre.

select

Este verbo permite que uno *seleccione* las variables con las que uno quiere trabajar. Algunas veces otras variables pueden incomodar o no ser necesarias.

```
personas %>% select(Nombre,Apellido,Edad)
```

```
##      Nombre Apellido Edad
## 1    Pablo Gonzalez   26
## 2     Ana   Solano   25
## 3 Fernando   Vargas   19
## 4    Maria    Solis   24
```

select

Este verbo permite que uno *seleccione* las variables con las que uno quiere trabajar. Algunas veces otras variables pueden incomodar o no ser necesarias.

```
personas %>% select(Nombre,Apellido,Edad)
```

```
##      Nombre Apellido Edad
## 1      Pablo Gonzalez   26
## 2        Ana   Solano   25
## 3 Fernando   Vargas   19
## 4      Maria    Solis   24
```

Es equivalente a usar:

```
personas %>% `[`(c('Nombre','Apellido','Edad'))
personas[c('Nombre','Apellido','Edad')]
```

Pero es bastante más cómodo de escribir.

Eliminar con select

Para quitar columnas de un dataframe se puede usar el - (menos), para indicar que esa columna no:

```
personas %>% select(-Edad,-Sexo,-Asegurado)
```

##		Nombre	Provincia	Antigüedad	Casado	Apellido
## 1		Pablo	San Jose	5	TRUE	Gonzalez
## 2		Ana	Cartago	3	TRUE	Solano
## 3		Fernando	Limon	1	TRUE	Vargas
## 4		Maria	Limon	2	FALSE	Solis

select

También se le pueden indicar *rangos* de la forma X:Y para que tome todas las variables desde X hasta Y:

```
personas %>% select(Provincia:Casado)
```

##	Provincia	Asegurado	Antigüedad	Casado
## 1	San Jose	TRUE	5	TRUE
## 2	Cartago	TRUE	3	TRUE
## 3	Limon	FALSE	1	TRUE
## 4	Limon	FALSE	2	FALSE

Seleccionar por numero de columna

Se pueden seleccionar columnas utilizando su posición en la tabla:

```
personas %>% select(3:6)
```

##		Sexo	Provincia	Asegurado	Antigüedad
## 1		M	San Jose	TRUE	5
## 2		F	Cartago	TRUE	3
## 3		M	Limon	FALSE	1
## 4		F	Limon	FALSE	2

Y se pueden combinar números con nombres:

```
personas %>% select(2:4,Casado)
```

##		Edad	Sexo	Provincia	Casado
## 1		26	M	San Jose	TRUE
## 2		25	F	Cartago	TRUE
## 3		10	M	Limon	TRUE

Más funciones

Hay una serie de funciones diseñadas para complementar `select`, y poder trabajar con mayor facilidad. Entre estas están:

- ▶ `starts_with('algo')` que busca las columnas que comienzan con `'algo'`

Más funciones

Hay una serie de funciones diseñadas para complementar `select`, y poder trabajar con mayor facilidad. Entre estas están:

- ▶ `starts_with('algo')` que busca las columnas que comienzan con `'algo'`
- ▶ `ends_with('mas')` que busca las columnas que terminan con `'mas'`

Más funciones

Hay una serie de funciones diseñadas para complementar `select`, y poder trabajar con mayor facilidad. Entre estas están:

- ▶ `starts_with('algo')` que busca las columnas que comienzan con 'algo'
- ▶ `ends_with('mas')` que busca las columnas que terminan con 'mas'
- ▶ `contains('otra')` que busca las columnas que contienen 'otra' en su nombre

Más funciones

Hay una serie de funciones diseñadas para complementar `select`, y poder trabajar con mayor facilidad. Entre estas están:

- ▶ `starts_with('algo')` que busca las columnas que comienzan con `'algo'`
- ▶ `ends_with('mas')` que busca las columnas que terminan con `'mas'`
- ▶ `contains('otra')` que busca las columnas que contienen `'otra'` en su nombre
- ▶ `matches(...)` que busca las columnas que coincidan con la expresión regular(ver `?regexp`) que se indique.

Más funciones

Hay una serie de funciones diseñadas para complementar `select`, y poder trabajar con mayor facilidad. Entre estas están:

- ▶ `starts_with('algo')` que busca las columnas que comienzan con 'algo'
- ▶ `ends_with('mas')` que busca las columnas que terminan con 'mas'
- ▶ `contains('otra')` que busca las columnas que contienen 'otra' en su nombre
- ▶ `matches(...)` que busca las columnas que coincidan con la expresión regular(ver `?regexp`) que se indique.
- ▶ `num_range('x', 1:15)` que busca las columnas que sean de la forma `x1, x2, ..., x15`

Más funciones

Hay una serie de funciones diseñadas para complementar `select`, y poder trabajar con mayor facilidad. Entre estas están:

- ▶ `starts_with('algo')` que busca las columnas que comienzan con 'algo'
- ▶ `ends_with('mas')` que busca las columnas que terminan con 'mas'
- ▶ `contains('otra')` que busca las columnas que contienen 'otra' en su nombre
- ▶ `matches(...)` que busca las columnas que coincidan con la expresión regular(ver `?regexp`) que se indique.
- ▶ `num_range('x', 1:15)` que busca las columnas que sean de la forma `x1, x2, ..., x15`

Más funciones

Hay una serie de funciones diseñadas para complementar `select`, y poder trabajar con mayor facilidad. Entre estas están:

- ▶ `starts_with('algo')` que busca las columnas que comienzan con 'algo'
- ▶ `ends_with('mas')` que busca las columnas que terminan con 'mas'
- ▶ `contains('otra')` que busca las columnas que contienen 'otra' en su nombre
- ▶ `matches(...)` que busca las columnas que coincidan con la expresión regular(ver `?regexp`) que se indique.
- ▶ `num_range('x',1:15)` que busca las columnas que sean de la forma `x1`, `x2`,...,`x15`

Seleccione todas las columnas que terminen con 'ado', junto con el Nombre de la persona.

Más funciones

Hay una serie de funciones diseñadas para complementar `select`, y poder trabajar con mayor facilidad. Entre estas están:

- ▶ `starts_with('algo')` que busca las columnas que comienzan con 'algo'
- ▶ `ends_with('mas')` que busca las columnas que terminan con 'mas'
- ▶ `contains('otra')` que busca las columnas que contienen 'otra' en su nombre
- ▶ `matches(...)` que busca las columnas que coincidan con la expresión regular(ver `?regexp`) que se indique.
- ▶ `num_range('x',1:15)` que busca las columnas que sean de la forma `x1, x2,...,x15`

Seleccione todas las columnas que terminen con 'ado', junto con el Nombre de la persona. Seleccione las columnas que comienzan con 'a'

Cambiar nombres

Con select se pueden cambiar los nombres de las variables (columnas) que se están considerando:

```
personas %>% select(Primer_Nombre = Nombre, Primer_Apellido = Apellido)
```

##	Primer_Nombre	Primer_Apellido
## 1	Pablo	Gonzalez
## 2	Ana	Solano
## 3	Fernando	Vargas
## 4	Maria	Solis

Pero tiene la desventaja de que se pierden las columnas que no se mencionen.

rename

Esto se puede arreglar de dos formas: incluir everything() en el llamado a select:

```
personas %>% select(Primer_Nombre = Nombre, Primer_Apellido = Apellido, everything())
```

O, se puede usar la función rename:

```
personas %>% rename(Primer_Nombre = Nombre, Primer_Apellido = Apellido) %>%  
  select(-Asegurado)
```

##	Primer_Nombre	Edad	Sexo	Provincia	Antigüedad	Casado	Primer_Apellido
## 1	Pablo	26	M	San Jose	5	TRUE	Gonzalez
## 2	Ana	25	F	Cartago	3	TRUE	Solano
## 3	Fernando	19	M	Limon	1	TRUE	Vargas
## 4	Maria	24	F	Limon	2	FALSE	Solis

rename

Esto se puede arreglar de dos formas: incluir everything() en el llamado a select:

```
personas %>% select(Primer_Nombre = Nombre, Primer_Apellido = Apellido, everything())
```

O, se puede usar la función rename:

```
personas %>% rename(Primer_Nombre = Nombre, Primer_Apellido = Apellido) %>%  
  select(-Asegurado)
```

##	Primer_Nombre	Edad	Sexo	Provincia	Antigüedad	Casado	Primer_Apellido
## 1	Pablo	26	M	San Jose	5	TRUE	Gonzalez
## 2	Ana	25	F	Cartago	3	TRUE	Solano
## 3	Fernando	19	M	Limon	1	TRUE	Vargas
## 4	Maria	24	F	Limon	2	FALSE	Solis

Cambie el nombre de la columna de Sexo a Genero.

rename

Esto se puede arreglar de dos formas: incluir everything() en el llamado a select:

```
personas %>% select(Primer_Nombre = Nombre, Primer_Apellido = Apellido, everything())
```

O, se puede usar la función rename:

```
personas %>% rename(Primer_Nombre = Nombre, Primer_Apellido = Apellido) %>%  
  select(-Asegurado)
```

##	Primer_Nombre	Edad	Sexo	Provincia	Antigüedad	Casado	Primer_Apellido
## 1	Pablo	26	M	San Jose	5	TRUE	Gonzalez
## 2	Ana	25	F	Cartago	3	TRUE	Solano
## 3	Fernando	19	M	Limon	1	TRUE	Vargas
## 4	Maria	24	F	Limon	2	FALSE	Solis

Cambie el nombre de la columna de Sexo a Genero.

mutate

mutate



Figure 2: Otro tipo de mutaciones

mutate

mutate

También se pueden realizar operaciones numéricas,

```
personas %>%  
  mutate(Edad_Inicio = Edad - Antigüedad) %>%  
  select(-Sexo,-Provincia) # para que quepa en la diapositiva
```

##		Nombre	Edad	Asegurado	Antigüedad	Casado	Apellido	Edad_Inicio
##	1	Pablo	26	TRUE	5	TRUE	Gonzalez	21
##	2	Ana	25	TRUE	3	TRUE	Solano	22
##	3	Fernando	19	FALSE	1	TRUE	Vargas	18
##	4	Maria	24	FALSE	2	FALSE	Solis	22

mutate

También se pueden realizar operaciones numéricas,

```
personas %>%  
  mutate(Edad_Inicio = Edad - Antigüedad) %>%  
  select(-Sexo,-Provincia) # para que quepa en la diapositiva
```

##		Nombre	Edad	Asegurado	Antigüedad	Casado	Apellido	Edad_Inicio
## 1		Pablo	26	TRUE	5	TRUE	Gonzalez	21
## 2		Ana	25	TRUE	3	TRUE	Solano	22
## 3		Fernando	19	FALSE	1	TRUE	Vargas	18
## 4		Maria	24	FALSE	2	FALSE	Solis	22

Si los años que le faltan a las personas para pensionarse es la diferencia entre 65 y su edad actual, ¿cuánto les falta para pensionarse?

mutate

También se pueden realizar operaciones numéricas,

```
personas %>%  
  mutate(Edad_Inicio = Edad - Antigüedad) %>%  
  select(-Sexo,-Provincia) # para que quepa en la diapositiva
```

##		Nombre	Edad	Asegurado	Antigüedad	Casado	Apellido	Edad_Inicio
## 1		Pablo	26	TRUE	5	TRUE	Gonzalez	21
## 2		Ana	25	TRUE	3	TRUE	Solano	22
## 3		Fernando	19	FALSE	1	TRUE	Vargas	18
## 4		Maria	24	FALSE	2	FALSE	Solis	22

Si los años que le faltan a las personas para pensionarse es la diferencia entre 65 y su edad actual, ¿cuánto les falta para pensionarse? ¿Cuanta antigüedad van a tener los 65 años? ¿Es más que 40 años?

mutate

También se pueden agregar columnas completamente nuevas con `mutate`, pero hay que tener cuidado de que tengan el orden correcto (arrange podría ser útil)

```
personas %>%  
  mutate(Bachillerato_en = c('Farmacia','Nutrición' ,  
                              'Actuariales','Economia'),  
         Hijos = c(FALSE,TRUE,FALSE,TRUE)) %>%  
  select(-(Edad:Antiguedad)) # tambien puedo eliminar columnas con rangos
```

##	Nombre	Casado	Apellido	Bachillerato_en	Hijos
## 1	Pablo	TRUE	Gonzalez	Farmacia	FALSE
## 2	Ana	TRUE	Solano	Nutrición	TRUE
## 3	Fernando	TRUE	Vargas	Actuariales	FALSE
## 4	Maria	FALSE	Solis	Economia	TRUE

mutate

También se pueden agregar columnas completamente nuevas con `mutate`, pero hay que tener cuidado de que tengan el orden correcto (arrange podría ser útil)

```
personas %>%  
  mutate(Bachillerato_en = c('Farmacia','Nutrición' ,  
                             'Actuariales','Economia'),  
         Hijos = c(FALSE,TRUE,FALSE,TRUE)) %>%  
  select(-(Edad:Antiguedad)) # tambien puedo eliminar columnas con rangos
```

##	Nombre	Casado	Apellido	Bachillerato_en	Hijos
## 1	Pablo	TRUE	Gonzalez	Farmacia	FALSE
## 2	Ana	TRUE	Solano	Nutrición	TRUE
## 3	Fernando	TRUE	Vargas	Actuariales	FALSE
## 4	Maria	FALSE	Solis	Economia	TRUE

En general se pueden agregar varias nuevas columnas.

mutate

Como en los demás verbos, se retorna un dataframe:

```
personas_act <- personas %>%  
  mutate(Bachillerato_en = c('Farmacia','Nutrición','Actuariales','Economía'),  
         Hijos = c(FALSE,TRUE,FALSE,TRUE)) %>%  
  mutate(Annos_pension = 65 - Edad,  
         Anno_pension_antiguedad = Antiguedad + 65 - Edad)  
str(personas_act)
```

```
## 'data.frame':      4 obs. of  12 variables:  
##  $ Nombre           : chr  "Pablo" "Ana" "Fernando" "Maria"  
##  $ Edad             : num  26 25 19 24  
##  $ Sexo             : chr  "M" "F" "M" "F"  
##  $ Provincia        : chr  "San Jose" "Cartago" "Limon" "Limon"  
##  $ Asegurado         : logi  TRUE TRUE FALSE FALSE  
##  $ Antiguedad        : num  5 3 1 2  
##  $ Casado            : logi  TRUE TRUE TRUE FALSE
```

mutate + ifelse

Una combinación muy útil es emplear mutate junto con ifelse, para revisar condiciones y luego **mutate** (se aclara la garganta) agrupar variables.

```
personas %>% mutate(Clasificacion_edad = ifelse(Edad<20,1,  
                                                ifelse(Edad>=25,3,2)))
```

##	Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
## 1	Pablo	26	M	San Jose	TRUE	5	TRUE	Gonzalez
## 2	Ana	25	F	Cartago	TRUE	3	TRUE	Solano
## 3	Fernando	19	M	Limon	FALSE	1	TRUE	Vargas
## 4	Maria	24	F	Limon	FALSE	2	FALSE	Solis
##	Clasificacion_edad							
## 1			3					
## 2			3					
## 3			1					
## 4			2					

mutate + otras funciones

También se pueden aplicar funciones vectoriales que retornen un único valor (sino, da problemas) para toda la tabla:

```
personas %>% mutate(Media_edad = mean(Edad),  
                     Max_antig = max(Antiguedad),  
                     Primer_Nombre = first(Nombre),  
                     Ultimo_Ap = last(Apellido),  
                     Cantidad = n()) %>%  
  select(-Sexo,-Provincia,-Casado)
```

##	Nombre	Edad	Asegurado	Antiguedad	Apellido	Media_edad	Max_antig
## 1	Pablo	26	TRUE	5	Gonzalez	23.5	5
## 2	Ana	25	TRUE	3	Solano	23.5	5
## 3	Fernando	19	FALSE	1	Vargas	23.5	5
## 4	Maria	24	FALSE	2	Solis	23.5	5
##	Primer_Nombre	Ultimo_Ap	Cantidad				
## 1	Pablo	Solis	4				

Por este tipo de funciones es que `arrange` es *taaaaaan* útil.

- ▶ Usando `arrange` y `mutate` indique cuanto es la diferencia de edades con la menor edad.

Por este tipo de funciones es que `arrange` es *taaaaaan* útil.

- ▶ Usando `arrange` y `mutate` indique cuanto es la diferencia de edades con la menor edad.
- ▶ Combinando `filter` y `mutate` indique cuantas personas hay casadas.

Por este tipo de funciones es que `arrange` es *taaaaaan* útil.

- ▶ Usando `arrange` y `mutate` indique cuanto es la diferencia de edades con la menor edad.
- ▶ Combinando `filter` y `mutate` indique cuantas personas hay casadas.
- ▶ Repita lo anterior, pero con la cantidad de personas que tienen sexo 'M'

Por este tipo de funciones es que `arrange` es *taaaaaan* útil.

- ▶ Usando `arrange` y `mutate` indique cuanto es la diferencia de edades con la menor edad.
- ▶ Combinando `filter` y `mutate` indique cuantas personas hay casadas.
- ▶ Repita lo anterior, pero con la cantidad de personas que tienen sexo 'M'
- ▶ Si ordena por edad y pega el último nombre con el primer apellido, ¿cuál sería el nombre completo?

transmute

Esta función hace lo mismo que mutate, con la excepción de que se quitan todas las columnas *viejas*:

```
personas %>%  
  transmute(Clasificacion_edad = ifelse(Edad<20,1,ifelse(Edad>=25,3,2)),  
            Dif_Antig = Antiguedad - first(Antiguedad),  
            Apellido,  
            Num_persona = row_number())
```

	Clasificacion_edad	Dif_Antig	Apellido	Num_persona
## 1	3	0	Gonzalez	1
## 2	3	-2	Solano	2
## 3	1	-4	Vargas	3
## 4	2	-3	Solis	4

Otras funciones:

Todas las funciones que devuelvan un vector con la misma (o menor a) cantidad de entradas de las que reciban* se pueden aplicar usando un `mutate`. Entre estas, están la mayoría de las de funciones de utilidad de la primera presentación, y:

Otras funciones:

Todas las funciones que devuelvan un vector con la misma (o menor a) cantidad de entradas de las que reciban* se pueden aplicar usando un `mutate`. Entre estas, están la mayoría de las de funciones de utilidad de la primera presentación, y:

función	resultado
<code>lag</code>	la entrada anterior
<code>lead</code>	la entrada siguiente
<code>cummean</code>	media acumulada
<code>cummin</code>	minimo acumulado
<code>row_number</code>	numero de fila
<code>ntile</code>	agrupa en n cajas
<code>case_when</code>	ifelse general

summarise

summarise

Resume la información con la función de resumen que se indique, como con mutate, de la forma que se indique.

summarise

Resume la información con la función de resumen que se indique, como con mutate, de la forma que se indique.

```
personas %>%  
  summarise(Total_personas = n(),  
            Max_edad = max(Edad),  
            Min_antig = min(Antiguedad))
```

```
##   Total_personas Max_edad Min_antig  
## 1                4       26        1
```

summarise

Resume la información con la función de resumen que se indique, como con mutate, de la forma que se indique.

```
personas %>%  
  summarise(Total_personas = n(),  
            Max_edad = max(Edad),  
            Min_antig = min(Antiguedad))
```

```
##   Total_personas Max_edad Min_antig  
## 1                4       26        1
```

Note que solo se obtiene una fila y se pierden todas las demás.

summarise

Resume la información con la función de resumen que se indique, como con mutate, de la forma que se indique.

```
personas %>%  
  summarise(Total_personas = n(),  
            Max_edad = max(Edad),  
            Min_antig = min(Antiguedad))
```

```
##   Total_personas Max_edad Min_antig  
## 1                4       26        1
```

Note que solo se obtiene una fila y se pierden todas las demás.

Usando filter, mutate y summarise indique cuanto es lo máximo que se separan las edades de las antigüedades de los hombres casados.

summarise

Resume la información con la función de resumen que se indique, como con mutate, de la forma que se indique.

```
personas %>%  
  summarise(Total_personas = n(),  
            Max_edad = max(Edad),  
            Min_antig = min(Antiguedad))
```

```
##   Total_personas Max_edad Min_antig  
## 1                4       26        1
```

Note que solo se obtiene una fila y se pierden todas las demás.

Usando filter, mutate y summarise indique cuanto es lo máximo que se separan las edades de las antigüedades de los hombres casados.

Por sí sola, la función de summarise es útil, pero saca todo su potencial cuando se combina con:

group_by

Definition of a Group



A group consists of a nonempty set G together with a binary operation \star , such that the following properties hold:

- (1) Closure: If $a, b \in G$, then $a \star b \in G$.
- (2) Associativity: For all $a, b, c \in G$, we have $a \star (b \star c) = (a \star b) \star c$.
- (3) Identity: There exists an element $e \in G$ such that, for any $a \in G$, $a \star e = e \star a = a$.
- (4) Inverses: For any $a \in G$, there exists an element $a^{-1} \in G$ such that $a \star a^{-1} = a^{-1} \star a = e$.

Example

$$\langle \mathbb{R}, + \rangle$$

$$\text{ID: } 0$$

$$\text{INV. of } a: -a$$

YES

Example

$$\langle \mathbb{R}^+, \cdot \rangle$$

$$\{x \in \mathbb{R}, x > 0\}$$

$$\text{ID: } 1$$

$$\text{INV. of } a: \frac{1}{a}$$

YES

Example

$$\langle \mathbb{Q}, + \rangle$$

$$\text{ID: } 0$$

$$\text{INV. of } a: -a$$

YES

Example

$$\langle \mathbb{Q}, \cdot \rangle$$

$$\text{ID: } 1$$

$$\text{INV. of } a: \frac{1}{a}$$

$$3 \quad \frac{1}{3}$$

$$0 \quad \frac{1}{0}$$

NO

Example

$$\langle \mathbb{Z}, \cdot \rangle$$

$$\text{ID: } 1$$

$$\text{INV. of } a: \frac{1}{a}$$

$$3 \quad \frac{1}{3}$$

NO

Figure 3: Otro tipo de grupos

group_by

group_by

```
personas %>%  
  group_by(Sexo) %>%  
  summarise(Total_por_sexo = n())
```

```
## # A tibble: 2 x 2  
##   Sexo Total_por_sexo  
##   <chr>          <int>  
## 1 F              2  
## 2 M              2
```

Basándose en esto, ¿qué cree que hace group_by?

group_by

```
personas %>%  
  group_by(Sexo) %>%  
  summarise(Total_por_sexo = n())
```

```
## # A tibble: 2 x 2  
##   Sexo Total_por_sexo  
##   <chr>          <int>  
## 1 F              2  
## 2 M              2
```

Basándose en esto, ¿qué cree que hace group_by?

Lo que hace group_by es crear “grupos” de tablas diferenciando por los posibles tipos de valores que toman los parámetros que se le indiquen, para que las operaciones siguientes que realice las haga como si fueran tablas separadas. Para *desagrupar* una tabla que está agrupada, se usa la función ungroup.

group_by + mutate

Al combinar `group_by` con `mutate`, se crea una columna nueva (como con `mutate` normal), pero sin perder las observaciones anteriores, como sí pasa con `summarise`:

group_by + mutate

Al combinar group_by con mutate, se crea una columna nueva (como con mutate normal), pero sin perder las observaciones anteriores, como sí pasa con summarise:

```
personas %>%  
  group_by(Sexo) %>% mutate(Total_sexo = n()) %>%  
  ungroup() %>% select(-(Provincia:Antiguedad))
```

```
## # A tibble: 4 x 6  
##   Nombre      Edad Sexo  Casado Apellido Total_sexo  
##   <chr>      <dbl> <chr> <lgl>   <chr>      <int>  
## 1 Pablo        26 M     TRUE   Gonzalez      2  
## 2 Ana          25 F     TRUE   Solano         2  
## 3 Fernando     19 M     TRUE   Vargas         2  
## 4 Maria        24 F     FALSE  Solis          2
```

Esto después se podría combinar con otro conteo más complejo, para estimar el porcentaje de personas que están en cada categoría. Y ya se tiene el porcentaje al que le corresponde cada persona.

Repita el ejercicio anterior, pero con Provincia.

Más de un grupo

Si se quiere agrupar con más de una columna, lo que se hace es que se incluye en los parámetros de `group_by`:

```
personas %>%  
  group_by(Sexo,Casado) %>%  
  summarise(Max_antig = max(Antiguedad))
```

```
## # A tibble: 3 x 3  
## # Groups:   Sexo [?]  
##   Sexo  Casado Max_antig  
##   <chr> <lgl>      <dbl>  
## 1 F     FALSE      2  
## 2 F     TRUE       3  
## 3 M     TRUE       5
```

Más de un grupo

Si se quiere agrupar con más de una columna, lo que se hace es que se incluye en los parámetros de `group_by`:

```
personas %>%  
  group_by(Sexo,Casado) %>%  
  summarise(Max_antig = max(Antiguedad))
```

```
## # A tibble: 3 x 3  
## # Groups:   Sexo [?]  
##   Sexo  Casado Max_antig  
##   <chr> <lgl>      <dbl>  
## 1 F      FALSE        2  
## 2 F      TRUE         3  
## 3 M      TRUE         5
```

¿Qué provincia tiene más personas aseguradas? Recuerde que `TRUE == 1`.

Más de un grupo

Si se quiere agrupar con más de una columna, lo que se hace es que se incluye en los parámetros de `group_by`:

```
personas %>%  
  group_by(Sexo,Casado) %>%  
  summarise(Max_antig = max(Antiguedad))
```

```
## # A tibble: 3 x 3  
## # Groups:   Sexo [?]  
##   Sexo  Casado Max_antig  
##   <chr> <lgl>      <dbl>  
## 1 F     FALSE      2  
## 2 F     TRUE       3  
## 3 M     TRUE       5
```

¿Qué provincia tiene más personas aseguradas? Recuerde que `TRUE == 1`. ¿Cual condición de aseguramiento tiene la mayor diversidad de provincias? Use: `n_distinct`.

group_by + filter

Si se quisieran conservar los grupos que tengan al menos cierta cantidad de observaciones, se puede hacer:

```
personas %>%  
  group_by(Provincia) %>%  
  filter(n()>1) %>%  
  ungroup()
```

```
## # A tibble: 2 x 8
```

##	Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido
##	<chr>	<dbl>	<chr>	<chr>	<lgl>	<dbl>	<lgl>	<chr>
## 1	Fernando	19	M	Limon	FALSE	1	TRUE	Vargas
## 2	Maria	24	F	Limon	FALSE	2	FALSE	Solis

group_by + filter

Si se quisieran conservar los grupos que tengan al menos cierta cantidad de observaciones, se puede hacer:

```
personas %>%  
  group_by(Provincia) %>%  
  filter(n()>1) %>%  
  ungroup()
```

```
## # A tibble: 2 x 8
```

##	Nombre	Edad	Sexo	Provincia	Asegurado	Antigüedad	Casado	Apellido	
##	<chr>	<dbl>	<chr>	<chr>	<lgl>		<dbl>	<lgl>	<chr>
## 1	Fernando	19	M	Limon	FALSE		1	TRUE	Vargas
## 2	Maria	24	F	Limon	FALSE		2	FALSE	Solis

Repita el ejercicio, pero conservando las provincias que tengan como edad mínima al menos 24.

rowwise

Esta función permite aplicar funciones fila por fila, sin tener interacción entre filas. Es particularmente útil cuando se trabaja con funciones propias, las cuales resumen la información que se está trabajando, pero se van a tener resultados distintos para cada fila.

Próximamente:

Uniones de tablas :)