**TIDYVERSE**

**Introducción por CHATGPT:**

La librería "tidyverse" es un conjunto de paquetes de R diseñados para trabajar juntos de manera coherente y facilitar el análisis de datos. Fue desarrollado por Hadley Wickham y su equipo en RStudio, y se ha convertido en una de las librerías más populares y útiles en el mundo del análisis de datos con R.

El "tidyverse" incluye una serie de paquetes como "ggplot2", "dplyr", "tidyr", "readr", "purrr", "stringr" y otros más. Cada uno de estos paquetes aborda un aspecto específico del análisis de datos, desde la manipulación y limpieza de datos hasta la visualización y modelado.

Lo que hace que el "tidyverse" sea tan útil es que los paquetes se integran bien entre sí y utilizan una sintaxis coherente y fácil de entender. También promueve la filosofía "tidy data", que se refiere a la organización de los datos en tablas bien estructuradas y fáciles de analizar.

En resumen, el "tidyverse" es una herramienta poderosa y útil para el análisis de datos en R, y su popularidad sigue en aumento debido a su facilidad de uso y eficacia.

**Introducción:**

Como vemos en la introducción anterior, la IA de chatGPT nos da una idea bastante acertada de lo que es el universo ordenado o el “tidyverse”, aquí es donde R realmente se desprende de otros lenguajes que se utilizan para el análisis de datos, los lenguajes como Julia y Python han estado copiando estos esquemas y funciones desde hace ya algunos años, por ejemplo los famosos Pandas de Python se copiaron literalmente del dplyr, al ser código libre, se pueden hacer este tipo de copias hacia otros lenguajes, tengo entendido que Julia actualmente esta trabajando en la adaptación de la misma librería para su código.

Hasta ahora hemos manipulado algunos vectores, reordenándolos y creando conjuntos mediante índices. Sin embargo, cuando entramos a análisis mucho mas avanzados, la unidad de medida para almacenar los datos no son los vectores sino los dataframes, o en su forma avanzada el tibble.

**¿Qué vamos a aprender?**

Aprenderemos a trabajar directamente con dataframes y tibbles, lo que nos va a facilitar enormemente la organización de nuestra información. Utilizaremos dataframes en la gran mayoría de casos. Nos vamos a centrar en un formato de datos especifico llamado “tidy” además de una colección especifica de paquetes que serán particularmente útiles para trabajar con datos ordenados.

**Iniciando**

Si aun no lo hemos realizado procederemos a instalar nuestra librería de tidyverse además de dslabs con el siguiente comando:

install.packages**(**"tidyverse"**)**

install.packages**(**"dslabs"**)**

Aprenderemos como implementar el enfoque tidyverse a lo largo del curso, antes de profundizar en los detalles veremos a continuación algunas de las funcionalidades mas utilizadas en el tidyverse.

Vamos a comenzar con el paquete dplyr para manipular datos y el paquete purr para trabajar con funciones. Tengamos en cuenta que tidyverse incluye varios paquetes mas que iremos viendo poco a poco, lo primero es presentar el concepto y luego demostraremos y practicaremos como trabajar los datos en estos formatos.

**Tidy Data**

Podemos decir que el concepto de “tidy data”, se refiere a unas tablas que se encuentran en un formato ordenado si cada fila representa una observación y las columnas representan las diferentes variables disponibles para cada una de estas observaciones. En este caso veremos un conjunto de datos de la librería dslabs, llamado murders que es un dataframe en formato tidy.

library**(**tidyverse**)**

library**(**dslabs**)**

data**(**murders**)**

head**(**murders**)**

Veremos algo parecido a esto:

**>** head**(**murders**)**

state abb region population total

1 Alabama AL South 4779736 135

2 Alaska AK West 710231 19

3 Arizona AZ West 6392017 232

4 Arkansas AR South 2915918 93

5 California CA West 37253956 1257

6 Colorado CO West 5029196 65

**>**

Vemos como cada fila representa un estado y cada una de las 5 columnas proporcionan variables diferentes en relación con los mismos: nombre, abreviatura, región, población y total que corresponde al No. De asesinatos por estado.

Vamos a considerar el siguiente ejemplo para poder ver información en otros formatos:

gapminder %>% filter**(**country **==** c**(**"Germany", "South Korea"**))** %>% select**(**c**(**country, year, fertility**))** %>% head**()**

Obtenemos algo parecido a esto:

**>** gapminder %>% filter**(**country **==** c**(**"Germany", "South Korea"**))** %>% select**(**c**(**country, year, fertility**))** %>% head**()**

country year fertility

1 Germany 1960 2.41

2 South Korea 1960 6.16

3 Germany 1962 2.47

4 South Korea 1962 5.79

5 Germany 1964 2.49

6 South Korea 1964 5.36

Este conjunto de datos ordenado proporciona tasas de fertilidad para dos países a lo largo de los años. Este es un conjunto de datos ordenado porque cada fila presenta una observación con las tres variables: país, año y tasa de fertilidad.

Sin embargo, este conjunto de datos originalmente vino en otro formato y se reformó para el paquete dslabs. Originalmente, los datos estaban en el siguiente formato:



Aquí el código para acceder desde R:

dat **<-** read\_csv**(**"data/sp\_dyn\_tfrt\_in.csv"**)**

dat %>% filter**(**country **==** "Germany"**)** %>% head**()**

Les pasaré el archivo o lo colgare en OneDrive para su descarga.

Se proporciona la misma información, pero hay dos diferencias importantes en el formato:

1. Cada fila incluye varias observaciones.
2. Una de las variables, año, se almacena en el encabezado.

Para que los paquetes tidyverse se utilicen de manera óptima, los datos deben volver a formarse en un formato ordenado (tidy), que aprenderá a hacer en la parte de domado de datos. Hasta entonces, utilizaremos conjuntos de datos de ejemplo que ya están en formato ordenado (tidy).

Aunque no es inmediatamente obvio, a medida que avance en el curso comenzará a apreciar las ventajas de trabajar en un marco en el que las funciones utilizan formatos ordenados tanto para las entradas como para las salidas. Verá cómo esto permite que el analista de datos se centre en aspectos más importantes del análisis en lugar del formato de los datos.

**DPLYR**

El paquete dplyr del tidyverse presenta funciones que realizan algunas de las operaciones más comunes cuando se trabaja con dataframes y usa nombres para estas funciones que son relativamente fáciles de recordar.

**Por ejemplo:**

* Para cambiar la tabla de datos agregando una nueva columna, usamos mutate().
* Para filtrar la tabla de datos a un subconjunto de filas, usamos filter().
* Finalmente, para subdividir los datos seleccionando columnas específicas, usamos select().

En este enlace podemos bajar la hoja de ayuda tanto para dplyr combinada con tidyr que se llama domado de datos o data Wrangling.

<https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf>

Función mutate()

Vamos a aprender a usar esta función de la librería dplyr, esta función nos permite agregar una columna extra con distintos cálculos, o llamando otras funciones.

Como ejemplo, vamos a querer que dentro del propio dataframe exista una columna extra en donde agregaremos la tasa de asesinatos por estado.

La función mutate() toma el dataframe como primer argumento y el nombre y los valores de la variable como segundo argumento usando la convención nombre = valores.

Entonces, para agregar tasas de asesinatos por cada 100,000 habitantes, usamos:

library**(**tidyverse**)**

library**(**dslabs**)**

data**(**"murders"**)**

murders **<-** mutate**(**murders, rate **=** total**/**population **\*** 10**^**5**)**

Aquí asignamos al objeto murders la opción de mutate, el primer argumento es el propio dataframe, el segundo es la operación de tasa que será igual al total de asesinatos dividido la población, la cual multiplicamos por 10 elevado a la 5 que es igual a 100,000.

Observe que aquí usamos total y población dentro de la función, que son objetos que no están definidos en nuestro espacio de trabajo. Pero ¿por qué no recibimos un error?

Esta es una de las principales características de dplyr. Las funciones en este paquete, como mutate, saben buscar variables en el dataframe proporcionado en el primer argumento. En la llamada a mutate() anterior, total tendrá los valores en murders$total. Este enfoque hace que el código sea mucho más legible.

Ahora observamos los cambios con la función head() y obtendremos algo como esto:

head**(**murders**)**

state abb region population total rate

1 Alabama AL South 4779736 135 2.824424

2 Alaska AK West 710231 19 2.675186

3 Arizona AZ West 6392017 232 3.629527

4 Arkansas AR South 2915918 93 3.189390

5 California CA West 37253956 1257 3.374138

6 Colorado CO West 5029196 65 1.292453

Observación:

Aunque hemos sobrescrito el objeto de asesinatos original, esto no cambia el objeto que se cargó con datos (murders). Si volvemos a cargar los datos murders, el original sobrescribirá nuestra versión mutada.

Función filter()

Ahora suponga que queremos filtrar la tabla de datos para mostrar solo las entradas para las cuales la tasa de homicidios es inferior a 0,71.

Para hacer esto, utilizamos la función filter(), que toma la tabla de datos como el primer argumento y luego la declaración condicional como el segundo. Al igual que mutate(), podemos usar los nombres de variables sin comillas de murders dentro de la función y sabrá que nos referimos a las columnas y no a los objetos en el espacio de trabajo.

filter**(**murders, rate **<=** 0.71**)**

state abb region population total rate

1 Hawaii HI West 1360301 7 0.5145920

2 Iowa IA North Central 3046355 21 0.6893484

3 New Hampshire NH Northeast 1316470 5 0.3798036

4 North Dakota ND North Central 672591 4 0.5947151

5 Vermont VT Northeast 625741 2 0.3196211

Función select()

Aunque nuestra tabla de datos solo tiene seis columnas, algunas tablas de datos incluyen cientos. Si queremos ver algunas, podemos usar la función dplyr select().

En el siguiente código seleccionamos tres columnas, asignamos esto a un nuevo objeto y luego filtramos el nuevo objeto:

new\_table **<-** select**(**murders, state, region, rate**)**

filter**(**new\_table, rate **<=** 0.71**)**

En la llamada a select(), el primer argumento de murders es un objeto en este caso un dataframe, pero estado, región y tasa son nombres de variables.

Nuestro resultado será muy similar a este:

**>** new\_table **<-** select**(**murders, state, region, rate**)**

**>** filter**(**new\_table, rate **<=** 0.71**)**

state region rate

1 Hawaii West 0.5145920

2 Iowa North Central 0.6893484

3 New Hampshire Northeast 0.3798036

4 North Dakota North Central 0.5947151

5 Vermont Northeast 0.3196211

Operadores de tubería (pipes)

Con dplyr podemos realizar una serie de operaciones, por ejemplo, seleccionar y luego filtrar, enviando los resultados de una función a otra y así sucesivamente utilizando lo que se llama operador de tubería o pipes en inglés: **%>%**.

Escribimos el código anterior para mostrar tres variables (estado, región, tasa) para los estados que tienen tasas de homicidio por debajo de 0,71. Para hacer esto, definimos el objeto intermedio new\_table. En dplyr podemos escribir código que se parezca más a una descripción de lo que queremos hacer sin objetos intermedios:

* original data → select → filter

Para tal operación, podemos usar la tubería%>% (pipes). El código se ve así:

library**(**tidyverse**)**

library**(**dslabs**)**

data**(**"murders"**)**

murders **<-** mutate**(**murders, rate **=** total**/**population **\*** 10**^**5**)**

murders %>% select**(**state, region, rate**)** %>% filter**(**rate **<=** 0.71**)**

Algunos se preguntarán si se puede hacer todo de un solo plumazo, o de corrido, por supuesto que, si se puede, sin embargo, aquí separe en dos líneas de código para hacerlo más comprensible.

Esta línea de código es equivalente a las dos líneas de código anteriores.

- ¿Que está pasando aqui?

En general, la tubería envía el resultado del lado izquierdo de la tubería como primer argumento de la función en el lado derecho de la tubería. Aquí hay un ejemplo muy simple:

16 %>% sqrt**()** %>% log**(**base **=** 2**)**

Aquí el 16 lo pasamos a su raíz cuadrada que será 4 y este resultado al log base 2 que es igual a dos, lo podemos partir para mejor comprensión así:

**>** 16 %>% sqrt**()**

**[**1**]** 4

**>** 4 %>% log**(**base **=** 2**)**

**[**1**]** 2

**>**

Por lo tanto, al usar la tubería con dataframe y dplyr, ya no necesitamos especificar el primer argumento requerido ya que las funciones de dplyr que hemos descrito toman todos los datos como el primer argumento. En el código que escribimos:

murders %>% select**(**state, region, rate**)** %>% filter**(**rate **<=** 0.71**)**

Tenga en cuenta que la tubería funciona bien con funciones donde el primer argumento son los datos de entrada. Las funciones en paquetes tidyverse como dplyr tienen este formato y se pueden usar fácilmente con la tubería.

**Resumiendo Datos**

Una parte importante del análisis exploratorio de datos es resumir los datos.

El promedio y la desviación estándar son dos ejemplos de estadísticas de resumen ampliamente utilizadas.

A menudo se pueden obtener resúmenes más informativos dividiendo primero los datos en grupos.

En esta sección, cubrimos dos nuevas funciones dplyr que facilitan estos cálculos:

* summarize()
* group\_by().

Aprendemos a acceder a los valores resultantes utilizando la función pull().

Función summarize()

La función summarize() en dplyr proporciona una forma de calcular estadísticas de resumen con código intuitivo y legible. Comenzamos con un ejemplo simple basado en alturas. El conjunto de datos de alturas incluye las alturas y el sexo reportados por los estudiantes en una encuesta en clase.

Aquí el código:

library**(**dplyr**)**

library**(**dslabs**)**

data**(**heights**)**

El siguiente código calcula el promedio y la desviación estándar para las mujeres:

s **<-** heights %>%

filter**(**sex **==** "Female"**)** %>%

summarize**(**promedio **=** mean**(**height**)**, desviacion\_standard **=** sd**(**height**))**

s

Esto toma nuestra tabla de datos original como entrada, la filtra para mantener solo a las mujeres y luego produce una nueva tabla resumida con solo el promedio y la desviación estándar de las alturas. Podemos elegir los nombres de las columnas de la tabla resultante. Por ejemplo, arriba decidimos usar el promedio y la desviación estándar, pero podríamos haber usado otros nombres de la misma manera.

Debido a que la tabla resultante almacenada en s es un dataframe, podemos acceder a los componentes con el descriptor de acceso $:

s**$**promedio

s**$**desviacion\_standard

Al igual que con la mayoría de las otras funciones de dplyr, summarize() es consciente de los nombres de las variables y podemos usarlas directamente.

Entonces, dentro de la llamada a la función de resumen, escribimos media (altura), la función accede a la columna con el nombre "altura" y luego calcula el promedio del vector numérico resultante. Podemos calcular cualquier otro resumen que opera en vectores y devuelve un solo valor.

Por ejemplo, podemos agregar las alturas mediana, mínima y máxima de esta manera:

heights %>%

filter**(**sex **==** "Female"**)** %>%

summarize**(**median **=** median**(**height**)**, minimum **=** min**(**height**)**, maximum **=** max**(**height**))**

Nuestro resultado será algo parecido a esto:

**>** heights %>%

**+** filter**(**sex **==** "Female"**)** %>%

**+** summarize**(**median **=** median**(**height**)**, minimum **=** min**(**height**)**, maximum **=** max**(**height**))**

median minimum maximum

1 64.98031 51 79

**>**

Podemos obtener estos tres valores con una sola línea utilizando la función de cuantiles:

Por ejemplo, cuantil (x, c (0,0.5,1)) devuelve el mínimo (percentil 0), la mediana (percentil 50) y el máximo (percentil 100) del vector x.

Anteriormente, si intentamos usar una función como esta que devuelve dos o más valores dentro de summarize() obteníamos un error, hoy eso ha cambiado:

heights %>%

filter**(**sex **==** "Female"**)** %>%

summarize**(**range **=** quantile**(**height, c**(**0, 0.5, 1**)))**

Nuestro resultado será parecido a esto:

**>** heights %>%

**+** filter**(**sex **==** "Female"**)** %>%

**+** summarize**(**range **=** quantile**(**height, c**(**0, 0.5, 1**)))**

range

1 51.00000

2 64.98031

3 79.00000

**>**

Para otro ejemplo de cómo podemos usar la función summarize(), calculemos la tasa promedio de asesinatos en los Estados Unidos. Recuerde que nuestra tabla de datos incluye asesinatos totales y el tamaño de la población para cada estado y ya hemos utilizado dplyr para agregar una columna de índice de asesinatos:

data**(**murders**)**

murders **<-** murders %>% mutate**(**rate **=** total**/**population**\***10**^**5**)**

Recuerde que la tasa de asesinatos en los Estados Unidos no es el promedio de las tasas de asesinatos estatales y el siguiente código estaría errado:

summarize**(**murders, mean**(**rate**))**

Esto se debe a que, en el cálculo anterior, los estados pequeños tienen el mismo peso que los grandes. La tasa de homicidios en los Estados Unidos es el número total de asesinatos en los Estados Unidos dividido por la población total de los Estados Unidos, entonces el cálculo correcto es:

us\_murder\_rate **<-** murders %>%

summarize**(**rate **=** sum**(**total**)** **/** sum**(**population**)** **\*** 10**^**5**)**

us\_murder\_rate

Este cálculo cuenta estados más grandes proporcionalmente a su tamaño, lo que da como resultado un valor mayor.

Función pull()

El objeto us\_murder\_rate definido anteriormente representa solo un número. Sin embargo, lo estamos almacenando en un dataframe:

class**(**us\_murder\_rate**)**

**>** class**(**us\_murder\_rate**)**

**[**1**]** "data.frame"

Dado que, como la mayoría de las funciones de dplyr, summarize siempre devolverá un dataframe.

Esto podría ser problemático si queremos usar este resultado con funciones que requieren un valor numérico. Aquí mostramos un truco útil para acceder a los valores almacenados en los datos cuando se usan tuberías:

Cuando un objeto de datos se canaliza a ese objeto y se puede acceder a sus columnas usando la función pull().

Para entender lo que queremos decir, eche un vistazo a esta línea de código:

us\_murder\_rate %>% pull**(**rate**)**

Esto devuelve el valor en la columna de tasa de us\_murder\_rate, por lo que es equivalente a us\_murder\_rate$rate.

Para obtener un número de la tabla de datos original con una línea de código, podemos escribir:

us\_murder\_rate **<-** murders %>%

summarize**(**rate **=** sum**(**total**)** **/** sum**(**population**)** **\*** 10**^**5**)** %>%

pull**(**rate**)**

us\_murder\_rate

Que ahora es numérico:

**>** class**(**us\_murder\_rate**)**

**[**1**]** "numeric"

**>**

Función group\_by()

Una operación común en la exploración de datos es dividir primero los datos en grupos y luego calcular resúmenes para cada grupo. Por ejemplo, es posible que queramos calcular la desviación promedio y estándar para las alturas de hombres y mujeres por separado. La función group\_by nos ayuda a hacer esto.

Si escribimos esto:

heights %>% group\_by**(**sex**)** %>% head**()**

Obtendremos esto:

**>** heights %>% group\_by**(**sex**)** %>% head**()**

# A tibble: 6 × 2

# Groups: sex [2]

sex height

**<**fct**>** **<**dbl**>**

1 Male 75

2 Male 70

3 Male 68

4 Male 74

5 Male 61

6 Female 65

El resultado no se ve muy diferente de las alturas, excepto que vemos Grupos: sexo cuando imprimimos el objeto. Aunque no es inmediatamente obvio por su apariencia, ahora es un dataframe especial llamado dataframe agrupado y las funciones de dplyr, en particular, se comportarán de manera diferente cuando actúen sobre este objeto.

Conceptualmente, puede pensar en esta tabla como muchas tablas, con las mismas columnas, pero no necesariamente el mismo número de filas, apiladas juntas en un objeto. Cuando resumimos los datos después de la agrupación, esto es lo que sucede:

heights %>%

group\_by**(**sex**)** %>%

summarize**(**average **=** mean**(**height**)**, standard\_deviation **=** sd**(**height**))**

**>** heights %>%

**+** group\_by**(**sex**)** %>%

**+** summarize**(**average **=** mean**(**height**)**, standard\_deviation **=** sd**(**height**))**

# A tibble: 2 × 3

sex average standard\_deviation

**<**fct**>** **<**dbl**>** **<**dbl**>**

1 Female 64.9 3.76

2 Male 69.3 3.61

La función summarize() aplica el resumen a cada grupo por separado.

Para otro ejemplo, calculemos la tasa media de asesinatos en las cuatro regiones del país:

murders %>%

group\_by**(**region**)** %>%

summarize**(**median\_rate **=** median**(**rate**))**

Pueden explorar ustedes mismos el resultado, que está vez no lo he colocado por acá.

**Ordenando Data Frames**

Al examinar un conjunto de datos, a menudo es conveniente ordenar la tabla por las diferentes columnas. Sabemos acerca de la función order() y sort(), pero para ordenar tablas enteras, la función de dplyr arrange() es útil. Por ejemplo, aquí ordenamos los estados por tamaño de población:

Función arrange()

library**(**tidyverse**)**

library**(**dslabs**)**

data**(**murders**)**

murders %>%

arrange**(**population**)** %>%

head**()**

Este es el resultado:

**>** murders %>%

**+** arrange**(**population**)** %>%

**+** head**()**

state abb region population total

1 Wyoming WY West 563626 5

2 District of Columbia DC South 601723 99

3 Vermont VT Northeast 625741 2

4 North Dakota ND North Central 672591 4

5 Alaska AK West 710231 19

6 South Dakota SD North Central 814180 8

Como vemos lo ha ordenado por población en orden ascendente.

Con la funcion arrange() podemos decidir por qué columna ordenar. Para ver los estados por población, de menor a mayor, organizamos por rate:

murders **<-** mutate**(**murders, rate **=** total**/**population **\*** 10**^**5**)**

murders %>%

arrange**(**rate**)** %>%

head**()**

Tenga en cuenta que el comportamiento predeterminado es ordenar en orden ascendente. En dplyr, la función desc() transforma un vector para que esté en orden descendente. Para ordenar la tabla en orden descendente, podemos escribir:

murders %>% arrange**(**desc**(**rate**))** %>% head**()**

En caso les de error es porque posiblemente llamamos a la data murders nuevamente, podemos incluir la tasa nuevamente así:

murders **<-** mutate**(**murders, rate **=** total**/**population **\*** 10**^**5**)**

Clasificación Anidada

Si estamos ordenando por una columna con ligaduras, podemos usar una segunda columna para romper la ligadura. Del mismo modo, se puede usar una tercera columna para romper los lazos entre la primera y la segunda, y así sucesivamente.

**Ejemplo:**

Aquí ordenamos por región, luego dentro de la región ordenamos por índice de asesinatos:

library**(**tidyverse**)**

library**(**dslabs**)**

data**(**"murders"**)**

murders **<-** mutate**(**murders, rate **=** total**/**population **\*** 10**^**5**)**

murders %>%

arrange**(**region, desc**(**rate**))** %>%

head**()**

Nuestro resultado será algo parecido a esto:

**>** murders %>%

**+** arrange**(**region, desc**(**rate**))** %>%

**+** head**()**

state abb region population total rate

1 Pennsylvania PA Northeast 12702379 457 3.597751

2 New Jersey NJ Northeast 8791894 246 2.798032

3 Connecticut CT Northeast 3574097 97 2.713972

4 New York NY Northeast 19378102 517 2.667960

5 Massachusetts MA Northeast 6547629 118 1.802179

6 Rhode Island RI Northeast 1052567 16 1.520093

**>**

Función top\_n()

En el código anterior, hemos utilizado la función head() para evitar que la página se llene con todo el conjunto de datos. Si queremos ver una proporción mayor, podemos usar la función top\_n. Esta función toma un dataframe como primer argumento, el número de filas para mostrar en el segundo y la variable para filtrar en el tercero. Aquí hay un ejemplo de cómo ver las 10 filas superiores:

murders %>% top\_n**(**10, rate**)**

**>** murders %>% top\_n**(**10, rate**)**

state abb region population total rate

1 Arizona AZ West 6392017 232 3.629527

2 Delaware DE South 897934 38 4.231937

3 District of Columbia DC South 601723 99 16.452753

4 Georgia GA South 9920000 376 3.790323

5 Louisiana LA South 4533372 351 7.742581

6 Maryland MD South 5773552 293 5.074866

7 Michigan MI North Central 9883640 413 4.178622

8 Mississippi MS South 2967297 120 4.044085

9 Missouri MO North Central 5988927 321 5.359892

10 South Carolina SC South 4625364 207 4.475323

**>**

**Tibble**

Cuando ya tenemos operaciones en datos usando (tidy), esté necesariamente se tiene que almacenar como dataframe.

library**(**tidyverse**)**

library**(**dslabs**)**

data**(**murders**)**

class**(**murders**)**

En algunas páginas anteriores, presentamos la función group\_by, que permite estratificar los datos antes de calcular las estadísticas de summarize().

Pero:

* ¿dónde se almacena la información del dataframe?

Tenga en cuenta que no hay columnas con esta información.

Pero, si observa de cerca la salida a continuación, verá la línea A tibble: 6 x 5.

murders %>% group\_by**(**region**)** %>% head**()**

Podemos aprender la clase del objeto devuelto usando:

murders %>% group\_by**(**region**)** %>% class**()**

El tbl, pronunciado tibble, es un tipo especial de dataframe. Las funciones group\_by() y summarize() siempre devuelven este tipo de dataframe.

La función group\_by() devuelve un tipo especial de tbl, grouped\_df; Diremos más sobre esto más adelante.

Por coherencia, los verbos de manipulación \_\_dplyr\_\_ (select, filter, mutate y arrange) conservan la clase de la entrada: si reciben un datafame regular, devuelven un dataframe regular, mientras que, si reciben un tibble, devuelven un tibble. Pero los tibbles son el formato preferido en tidyverse y como resultado, las funciones tidyverse que producen un dataframe desde cero devuelven un tibble.

**Diferencias entre un tibble y un dataframe, acorde a chatGPT.**

En R, un "data frame" y un "tibble" son estructuras de datos muy similares, pero hay algunas diferencias notables:

Biblioteca requerida: los tibbles son parte del paquete tidyverse, por lo que es necesario cargar ese paquete para usar tibbles. Los data frames son parte de la base de R y se cargan automáticamente cuando se inicia R.

Salida por defecto: los tibbles imprimen solo las primeras 10 filas y todas las columnas que quepan en la pantalla, mientras que los data frames imprimen todas las filas y columnas.

Comportamiento del operador $: cuando se usa el operador "$" para acceder a una columna en un tibble, el resultado es siempre otro tibble. En cambio, en un data frame, el resultado puede ser un vector, una matriz o un data frame.

Manejo de datos faltantes: los tibbles tienen un comportamiento ligeramente diferente en cuanto al manejo de datos faltantes. Por ejemplo, en un tibble, si se realiza una operación aritmética que involucre un valor faltante, el resultado será un valor faltante. En un data frame, el resultado puede ser NA o NaN dependiendo de la operación.

En resumen, los tibbles son una versión moderna de los data frames en R, que ofrecen algunas mejoras en la usabilidad y el manejo de datos faltantes. Sin embargo, en la mayoría de los casos, los data frames siguen siendo una opción perfectamente válida y útil para la mayoría de los usuarios de R.

**Continuación…**

Los tibbles son muy similares a los dataframes. De hecho, puede pensar en ellos como una versión moderna de ellos. No obstante, hay tres diferencias importantes que describiremos.

* Los tibbles se ven mejor
* El método de impresión para tibbles es más legible que el dataframe. Para ver esto, compare los resultados de los asesinatos normal o si lo convertimos en un tibble. Podemos hacer esto usando as\_tibble(murders). Si usa RStudio, la salida para un tibble se ajusta al tamaño de su ventana.
* Los subset en un tibble son tibbles.

Continuando con nuestro ejemplo podemos recuperar un objeto que no es un dataframe, en este caso pondremos un vector como ejemplo:

class**(**murders**[**,4**])**

Aquí vemos que no es un dataframe, sino un vector numérico, siendo un tibble esto no nos pasará.

class**(**as\_tibble**(**murders**)[**,4**])**

Esto es útil en tidyverse ya que las funciones en su mayoria requieren dataframes como entrada.

Con tibbles, si desea acceder al vector que define una columna y no recuperar un dataframe, debe usar el descriptor de acceso $:

class**(**as\_tibble**(**murders**)$**population**)**

Una característica relacionada es que tibbles le dará una advertencia si intenta acceder a una columna que no existe. Si escribimos accidentalmente Population en lugar de population:

as\_tibble**(**murders**)$**Population

Los tibbles pueden tener entradas complejas.

Si bien las columnas del dataframe deben ser vectores de números, cadenas o valores lógicos, los tibbles pueden tener objetos más complejos, como listas o funciones.

* Además, podemos crear tibbles con funciones:

tibble**(**id **=** c**(**1, 2, 3**)**, func **=** c**(**mean, median, sd**))**

Los tibbles pueden ser agrupados.

La función group\_by devuelve un tipo especial de tibble: un tibble agrupado. Esta clase almacena información que le permite saber qué filas están en qué grupos.

Las funciones tidyverse, en particular la función de summarize(), conocen la información del grupo.

Cree un tibble usando tibble() en lugar de data.frame

A veces es útil para nosotros crear nuestros propios dataframes. Para crear un dataframe en formato tibble, puede hacerlo utilizando la función tibble().

grades **<-** tibble**(**names **=** c**(**"John", "Juan", "Jean", "Yao"**)**,

exam\_1 **=** c**(**95, 80, 90, 85**)**,

exam\_2 **=** c**(**90, 85, 85, 90**))**

grades

Tenga en cuenta que la base R (sin paquetes cargados) tiene una función con un nombre muy similar, data.frame, que puede usarse para crear un dataframe regular en lugar de un tibble.

grades **<-** data.frame**(**names **=** c**(**"John", "Juan", "Jean", "Yao"**)**,

exam\_1 **=** c**(**95, 80, 90, 85**)**,

exam\_2 **=** c**(**90, 85, 85, 90**))**

class**(**grades**$**names**)**

Para convertir un marco de datos normal en un tibble, puede usar la función as\_tibble.

as\_tibble**(**grades**)** %>% class**()**

Haciendo una observación importante, hay librerías de R que no trabajan muy bien con los tibbles, en especial algunas de estadística para sacar algunos modelos, siempre que estemos escribiendo el código y necesitemos una librería adicional para ayudarnos con nuestra tarea, una parte primordial es si nuestros datos son tibbles o dataframes, siempre hay que hacer algunas validaciones antes de trabajar con tibbles si usamos librerías fuera del universo ordenado (tidyverse).

Operador de punto.

Una de las ventajas de usar la tubería%>% es que no tenemos que seguir nombrando nuevos objetos mientras manipulamos el marco de datos. Como recordatorio rápido, si queremos calcular la tasa media de asesinatos para los estados del sur, en lugar de escribir:

library**(**tidyverse**)**

library**(**dslabs**)**

data**(**"murders"**)**

tab\_1 **<-** filter**(**murders, region **==** "South"**)**

tab\_2 **<-** mutate**(**tab\_1, rate **=** total **/** population **\*** 10**^**5**)**

rates **<-** tab\_2**$**rate

median**(**rates**)**

Podemos evitar definir nuevos objetos intermedios escribiendo en su lugar:

filter**(**murders, region **==** "South"**)** %>%

mutate**(**rate **=** total **/** population **\*** 10**^**5**)** %>%

summarize**(**median **=** median**(**rate**))** %>%

pull**(**median**)**

Podemos hacer esto porque cada una de estas funciones toma un dataframe como primer argumento. Pero ¿qué pasa si queremos acceder a un componente del dataframe?

* Por ejemplo, ¿qué pasa si la función pull() no está disponible y queremos acceder a tab\_2$rate?
* ¿Qué nombre de dataframe usaríamos?

Para esto, podemos usar el operador de puntos como respuesta.

Por ejemplo, para acceder al vector rate sin la función pull(), podríamos usar:

rates **<-**filter**(**murders, region **==** "South"**)** %>%

mutate**(**rate **=** total **/** population **\*** 10**^**5**)** %>%

.**$**rate

median**(**rates**)**

Las funciones tidyverse saben interpretar tibbles agrupados.

Además, para facilitar los comandos de encadenamiento a través de la tubería%>%, las funciones tidyverse devuelven constantemente los dataframes, ya que esto asegura que la salida de una función sea aceptada como la entrada de otra.

**PURRR**

**Introducción por chatGPT:**

Purrr es un paquete de R diseñado para ayudar en el proceso de programación funcional. Este paquete proporciona una serie de herramientas que permiten a los usuarios aplicar funciones de manera iterativa y manejar datos en estructuras complejas. El objetivo principal de Purrr es facilitar la tarea de trabajar con estructuras de datos en R, permitiendo un flujo de trabajo más eficiente y expresivo.

Entre las herramientas más destacadas de Purrr se encuentran las funciones map(), reduce() y walk(), que permiten aplicar una función a cada elemento de una lista o vector, reducir una lista a un solo valor y aplicar una función sin retorno a cada elemento de una lista, respectivamente. Además, Purrr incluye una serie de otras funciones que facilitan el trabajo con estructuras de datos complejas, como las funciones flatten(), transpose() y set\_names().

En resumen, Purrr es una herramienta muy útil para programadores y analistas de datos que trabajan con estructuras de datos complejas en R, ya que permite una programación más eficiente y expresiva, simplificando así el trabajo con datos.

**Introducción.**

Como les comenté en un inicio, R tiene mas de 25,000 paquetes para poder operar lo que necesitamos, sin embargo, nuestro objetivo es el análisis de datos eficiente y rápido, es por eso por lo que he estado buscando y vamos a utilizar paquetes que se usan por experiencia en el análisis exploratorio de datos. Sin embargo, esto solo es la punta del iceberg, queda de nuestra cuenta buscar más librerías que puedan apoyar de forma eficiente nuestro trabajo y el objetivo que tiene al final de cuentas.

Si realizaron correctamente los ejercicios anteriores, me imagino que se toparon ya con las funciones apply de la base de R, la cual nos permitió aplicar la misma función a cada elemento de un vector.

Vamos a comenzar con unos pequeños ejemplos, pero primero construiremos una función para ver su aplicación.

compute\_s\_n **<-** **function(**n**){**

x **<-** 1**:**n

sum**(**x**)**

**}**

Lo único que hará esta función es sumar todos los números iniciando del 1, hasta el que nosotros le coloquemos.

Vamos a usar sapply para poder calcular la suma de los primeros n enteros para varios valores de n, así:

n **<-** 1**:**25

s\_n **<-** sapply**(**n, compute\_s\_n**)**

s\_n

Este tipo de operación, que aplica la misma función o procedimiento a elementos de un objeto, es bastante común en el análisis de datos. El paquete purrr incluye funciones similares a sapply pero que interactúan mejor con otras funciones tidyverse. La principal ventaja es que podemos controlar mejor el tipo de funciones de salida.

Por el contrario, sapply puede devolver varios tipos de objetos diferentes; por ejemplo, podríamos esperar un resultado numérico de una línea de código, pero sapply podría convertir nuestro resultado en carácter en algunas circunstancias.

Las funciones de purrr nunca harán esto: devolverán objetos de un tipo específico o devolverán un error si esto no es posible.

La primera función de purrr que aprenderemos es map(), que funciona de manera muy similar a sapply pero siempre, sin excepción, devuelve una lista:

library**(**tidyverse**)**

s\_n **<-** map**(**n, compute\_s\_n**)**

class**(**s\_n**)**

s\_n

Recordemos que la librería de purrr es parte del tidyverse, por eso no es necesario cargarla aparte.

Si queremos un vector numérico, podemos usar map\_dbl que siempre devuelve un vector de valores numéricos.

s\_n **<-** map\_dbl**(**n, compute\_s\_n**)**

class**(**s\_n**)**

Esto produce los mismos resultados que la llamada sapply que se muestra arriba.

Función map\_df()

Una función de purrr particularmente útil para interactuar con el resto del tidyverse es map\_df, que siempre devuelve un tibble. Sin embargo, la función que se llama debe devolver un vector a o una lista con nombres. Por esta razón, el siguiente código daría como resultado un Argumento 1 que debe tener un error de nombre:

s\_n **<-** map\_df**(**n, compute\_s\_n**)**

En este caso necesitamos cambiar la función para que trabaje correctamente así:

compute\_s\_n **<-** **function(**n**){**

x **<-** 1**:**n

tibble**(**sum **=** sum**(**x**))**

**}**

s\_n **<-** map\_df**(**n, compute\_s\_n**)**

head**(**s\_n**)**

Función Reduce()

Permite reducir una lista o un vector a un solo valor aplicando una función de acumulación a cada elemento. Es similar a la función Reduce() de R base, pero con algunas diferencias importantes.

Supongamos que queremos calcular el producto de todos los elementos en un vector. En lugar de hacerlo de forma iterativa, podemos usar la función reduce() para hacerlo de manera más concisa:

# Creamos un vector de números

numeros **<-** c**(**2, 4, 6, 8, 10**)**

# Usamos la función reduce para calcular el producto

producto **<-** reduce**(**numeros, `**\***`**)**

producto

En este caso, la función reduce() toma el vector números y aplica la operación de multiplicación \* de forma iterativa a cada elemento de la lista. Al final, el resultado es el producto de todos los elementos del vector, que en este caso es 3840.

Es importante destacar que la función reduce() también puede ser utilizada con otros tipos de datos, no solamente vectores numéricos. Además, es posible especificar una función de acumulación personalizada como argumento en lugar de la operación de multiplicación \*.

Función walk()

La función walk() del paquete Purrr de R es una herramienta muy útil para aplicar una función sin retorno a cada elemento de una lista o vector. En lugar de crear una nueva lista o vector, como lo hace la función map(), la función walk() simplemente aplica la función a cada elemento y descarta el resultado.

Esto es útil cuando queremos realizar alguna operación o manipulación en los datos sin necesidad de almacenar los resultados.

Ejemplo:

Supongamos que tenemos una lista de números y queremos imprimir cada uno de ellos en la consola.

En lugar de hacer un bucle for o aplicar la función print() a cada elemento de la lista, podemos usar la función walk() para hacerlo de manera más concisa:

# Creamos una lista de números

numeros **<-** list**(**2, 4, 6, 8, 10**)**

# Usamos la función walk para imprimir cada elemento en la consola

walk**(**numeros, print**)**

En este caso, la función walk() toma la lista numeros y aplica la función print() a cada elemento de forma iterativa. Al final, se imprime cada número en la consola.

Es importante destacar que la función walk() no devuelve ningún valor, por lo que es útil cuando queremos realizar alguna operación o manipulación en los datos sin necesidad de almacenar los resultados. Además, es posible especificar una función personalizada como argumento en lugar de la función print().

Función Case\_When()

Esta función se encuentra en la librería dplyr, es útil para vectorizar sentencias condicionales. Es similar a ifelse pero puede generar cualquier número de valores, en lugar de solo VERDADERO o FALSO. Aquí hay un ejemplo que divide los números en negativos, positivos y 0:

library**(**tidyverse**)**

library**(**dslabs**)**

x **<-** c**(-**2, **-**1, 0, 1, 2**)**

case\_when**(**x **<** 0 **~** "Negative", x **>** 0 **~** "Positive", **TRUE** **~** "Zero"**)**

Un uso común para esta función es definir variables categóricas basadas en variables existentes. Por ejemplo, supongamos que queremos comparar las tasas de asesinatos en tres grupos de estados: Nueva Inglaterra, West Coast, South y otros. Para cada estado, debemos preguntar si está en Nueva Inglaterra, si no es así, preguntamos si está en la costa oeste, si no, preguntamos si está en el sur y si no, asignamos otro.

Así es como usamos case\_when para hacer esto:

murders %>%

mutate**(**group **=** case\_when**(**

abb %in% c**(**"ME", "NH", "VT", "MA", "RI", "CT"**)** **~** "New England",

abb %in% c**(**"WA", "OR", "CA"**)** **~** "West Coast",

region **==** "South" **~** "South",

**TRUE** **~** "other"**))** %>%

group\_by**(**group**)** %>%

summarize**(**rate **=** sum**(**total**)** **/** sum**(**population**)** **\*** 10**^**5**)** %>%

arrange**(**rate**)**

Función beetween()

Una operación común en el análisis de datos es determinar si un valor cae dentro de un intervalo. Podemos verificar esto usando condicionales. Por ejemplo, para verificar si los elementos de un vector x están entre a y b, podemos escribir

x >= a & x <= b

Sin embargo, esto puede volverse engorroso, especialmente dentro del enfoque tidyverse. La función between realiza la misma operación.

between(x, a, b)

**Área Practica**

En esta área vamos a encontrar una batería de ejercicios que nos ayudaran a repasar todo el contenido de este documento.

Es muy importante que sepamos contestarlos correctamente, el otro modulo es domado de datos, este es mucho mas exigente y toma en cuenta que manejen lo anterior de cabo a rabo.

Como las practicas anteriores, pueden crear grupos para resolver esta práctica, lo importante es comprenderla.

**Practica No.1**

Examine el conjunto de datos integrado “co2”.

Cuál de los siguientes es verdadero:

1. co2 son datos ordenados: tiene un año para cada fila.
2. co2 no está ordenado: necesitamos al menos una columna con un vector de caracteres.
3. co2 no está ordenado: es una matriz en lugar de un marco de datos.
4. co2 no está ordenado: para estar ordenado tendríamos que arreglarlo para que tenga tres columnas (año, mes y valor), luego cada observación de co2 tendría una fila.

**Practica No.2**

Examine el conjunto de datos integrado “ChickWeight”.

Cuál de los siguientes es verdadero:

1. ChickWeight no es ordenado: cada pollito tiene más de una fila.
2. ChickWeight es ordenado: cada observación (un peso) está representada por una fila. El pollito del que procede esta medida es una de las variables.
3. ChickWeight no está ordenado: nos falta la columna del año.
4. ChickWeight está ordenado: se almacena en un marco de datos.

**Practica No.3**

Examine el conjunto de datos integrado “BOD”.

Cuál de los siguientes es verdadero:

1. BOD no está ordenado: solo tiene seis filas.
2. BOD no está ordenado: la primera columna es solo un índice.
3. BOD está ordenado: cada fila es una observación con dos valores (tiempo y demanda)
4. BOD está ordenado: todos los conjuntos de datos pequeños están ordenados por definición.

**Practica No.4**

¿Cuál de los siguientes conjuntos de datos integrados está ordenado (puede elegir más de uno):

1. BJsales
2. EuStockMarkets
3. DNase
4. Formaldehyde
5. Orange
6. UCBAdmissions

**Practica No.5**

Cargue el paquete dplyr y el conjunto de datos de asesinatos (murders).

library(dplyr)

library(dslabs)

data(murders)

Puede agregar columnas usando la función dplyr mutate(). Esta función es consciente de los nombres de las columnas y dentro de la función puede llamarlos sin comillas:

murders <- mutate(murders, population\_in\_millions = population / 10^6)

1. Utilice la función mutate() para agregar una columna de asesinatos denominada tasa con la tasa de asesinatos por cada 100 000 como en el código de ejemplo anterior.

Asegúrese de redefinir los asesinatos como se hizo en el código de ejemplo anterior (asesinatos <- [su código]) para que podamos seguir usando esta variable.

Si rank(x) te da los rangos de x de menor a mayor, rank(-x) te da los rangos de mayor a menor.

1. Use la función mutate() para agregar un rango de columna que contenga el rango, de mayor a menor índice de homicidios. Asegúrate de redefinir los asesinatos para que podamos seguir usando esta variable.

Con **dplyr**, podemos usar select para mostrar solo ciertas columnas. Por ejemplo, con este código solo mostraríamos los estados y el tamaño de la población:

select(murders, state, population) |> head()

1. Utilice select() para mostrar los nombres de los estados y las abreviaturas en los asesinatos. No redefina los asesinatos, solo muestre los resultados.

La función filter() de **dplyr** se usa para elegir filas específicas del marco de datos para mantener. A diferencia de seleccionar, que es para columnas, el filtro es para filas.

Por ejemplo, puede mostrar solo la fila de Nueva York de esta manera:

filter(murders, state == "New York")

Puede usar otros vectores lógicos para filtrar filas.

1. Use filter() para mostrar los 5 estados principales con las tasas de homicidios más altas. Después de agregar la tasa de asesinatos y el rango, no cambie el conjunto de datos de asesinatos, solo muestre el resultado. Recuerde que puede filtrar según la columna de clasificación.

Podemos eliminar filas usando el operador!=. Por ejemplo, para eliminar Florida, haríamos esto:

no\_florida <- filter(murders, state != "Florida")

1. Cree un nuevo dataframe llamado no\_south que elimine estados de la región Sur.

¿Cuántos estados hay en esta categoría?

Puede usar la función nrow para esto.

También podemos usar %in% para filtrar con dplyr. Por lo tanto, puede ver los datos de Nueva York y Texas así:

filter(murders, state %in% c("New York", "Texas"))

1. Cree un nuevo dataframe llamado assassins\_nw con solo los estados del noreste y el oeste.

¿Cuántos estados hay en esta categoría?

Suponga que desea vivir en el noreste o el oeste y desea que la tasa de homicidios sea inferior a 1. Queremos ver los datos de los estados que satisfacen estas opciones. Tenga en cuenta que puede usar operadores lógicos con filtro. Aquí hay un ejemplo en el que filtramos para mantener solo los estados pequeños en la región noreste.

filter(murders, population < 5000000 & region == "Northeast")

Asegúrese de que los asesinatos se hayan definido con tasa y rango y aún tenga todos los estados.

1. Cree una tabla llamada my\_states que contenga filas para los estados que cumplan ambas condiciones: está en el noreste o el oeste y la tasa de homicidios es inferior a 1. Use select para mostrar solo el nombre del estado, la tasa y el rango.

**Practica No.6**

La tubería %>% se puede utilizar para realizar operaciones secuencialmente sin tener que definir objetos intermedios. Comience por redefinir el asesinato para incluir la tasa y el rango.

murders <- mutate(murders, rate = total / population \* 100000,

rank = rank(-rate))

En la solución del ejercicio anterior, hicimos lo siguiente:

my\_states <- filter(murders, region %in% c("Northeast", "West") &

rate < 1)

select(my\_states, state, rate, rank)

La tubería %>% nos permite realizar ambas operaciones secuencialmente sin tener que definir una variable intermedia my\_states. Por lo tanto, podríamos haber mutado y seleccionado en la misma línea así:

mutate(murders, rate = total / population \* 100000,

rank = rank(-rate)) %>%

select(state, rate, rank)

Observe que select ya no tiene un dataframe como primer argumento.

Se supone que el primer argumento es el resultado de la operación realizada justo antes de %>%.

1. Repita el ejercicio anterior, pero ahora en lugar de crear un nuevo objeto, muestre el resultado e incluya solo las columnas de estado, tasa y rango. Use una tubería %>% para hacer esto en una sola línea.

Restablezca los asesinatos a la tabla original mediante el uso de data(murders).

1. Use una canalización para crear un nuevo dataframe llamado my\_states que considere solo los estados en el noreste o el oeste que tienen una tasa de homicidios inferior a 1, y que contenga solo las columnas de estado, tasa y rango.

La tubería también debe tener cuatro componentes separados por tres %>% . El código debería ser algo como esto:

my\_states <- murders %>%

mutate SOMETHING %>%

filter SOMETHING %>%

select SOMETHING

**Practica No.7**

Para estos ejercicios, utilizaremos los datos de la encuesta recopilados por el Centro Nacional de Estadísticas de Salud de los Estados Unidos (NCHS).

Este centro ha realizado una serie de encuestas de salud y nutrición desde la década de 1960. A partir de 1999, se entrevistó a unas 5.000 personas de todas las edades cada año y completaron el componente de examen de salud de la encuesta.

Parte de los datos está disponible a través del paquete NHANES. Una vez que instale el paquete NHANES, puede cargar los datos de esta manera:

library(NHANES)

data(NHANES)

Los datos de NHANES tienen muchos valores faltantes. Las funciones mean y sd en R devolverán NA si alguna de las entradas del vector de entrada es NA. Aquí hay un ejemplo:

library(dslabs)

data(na\_example)

mean(na\_example)

*#> [1] NA*

sd(na\_example)

*#> [1] NA*

Para ignorarlo podemos usar el argumento na.rm

mean(na\_example, na.rm = TRUE)

*#> [1] 2.3*

sd(na\_example, na.rm = TRUE)

*#> [1] 1.22*

Exploremos ahora los datos de NHANES.

1. Proporcionaremos algunos datos básicos sobre la presión arterial. Primero, seleccionemos un grupo para establecer el estándar. Utilizaremos mujeres de 20 a 29 años. AgeDecade es una variable categórica con estas edades. Tenga en cuenta que la categoría está codificada como "20-29", ¡con un espacio al frente! ¿Cuál es el promedio y la desviación estándar de la presión arterial sistólica guardada en la variable BPSysAve? Guárdelo en una variable llamada ref.

Sugerencia: use filter(), summarize() y use el argumento na.rm = TRUE cuando calcule el promedio y la desviación estándar. También puede filtrar los valores NA usando filter().

1. Usando una tubería %>%, asigne el promedio a una variable numérica ref\_avg.

Sugerencia: use el código similar al anterior y luego pull().

1. Ahora reporte los valores mínimo y máximo para el mismo grupo.
2. Calcule el promedio y la desviación estándar para las mujeres, pero para cada grupo de edad por separado en lugar de una década seleccionada como en la pregunta 1. Tenga en cuenta que los grupos de edad están definidos por AgeDecade. Sugerencia: en lugar de filtrar por edad y género, filtre por Género y luego use group\_by.
3. Repita el ejercicio 4 para los hombres.
4. De hecho, podemos combinar ambos resúmenes de los ejercicios 4 y 5 en una sola línea de código. Esto se debe a que group\_by nos permite agrupar por más de una variable. Obtenga una gran tabla de resumen usando group\_by(AgeDecade, Gender).
5. Para hombres entre 40 y 49 años, compare la presión arterial sistólica entre razas según lo informado en la variable Raza1. Ordene la tabla resultante de menor a mayor presión arterial sistólica promedio.

**Practica No.8**

1. Cargue el conjunto de datos murders. ¿Cuál de los siguientes es verdadero?
2. murders está en formato ordenado y se almacena en un tibble.
3. murders está en formato ordenado y se almacena en un dataframe.
4. murders no está en formato ordenado y se almacena en un tibble.
5. murders no está en un formato ordenado y se almacena en un dataframe.
6. Use as\_tibble para convertir la tabla de datos murders en un tibble y guárdelo en un objeto llamado assassins\_tibble.
7. Utilice la función group\_by para convertir los asesinatos en un tibble agrupado por región.
8. Escriba un código tidyverse que sea equivalente a este código:

exp(media(log(murders$population)))

Escríbalo usando la tubería para que cada función se llame sin argumentos. Utilice el operador punto para acceder a la población. Sugerencia: el código debe comenzar con asesinatos %>%.

1. Utilice map\_df para crear un marco de datos con tres columnas denominadas n, s\_n y s\_n\_2. La primera columna debe contener los números del 1 al 100. La segunda y la tercera columna deben contener cada una la suma de 1 a n con n el número de fila.