Temario de la sesión

1. Repaso breve del curso básico
    1.1. Sintaxis general de R
    1.2. Estructuras básicas (vectors, data.frames, lists)
    1.3. Operadores lógicos
    1.4. Manejo de valores especiales: NaN, NULL, NA
2. Manejo avanzado de data frames
    2.1. Indexación múltiple y condicional
    2.2. Operaciones por columna y fila
    2.3. Filtrado con múltiples condiciones y operadores
3. Uso de dplyr
    3.1. Operaciones encadenadas para flujo de trabajo reproducible
    3.2. Verbos combinados: mutate(), group_by(), summarise(), arrange()
    3.3. Limpieza de datos reales con filter() y case_when()
    3.4. Uso de pivot_longer() y pivot_wider() desde tidyr
4. Introducción a funciones propias y programación modular
    4.1. Sintaxis general de funciones (function())
    4.2. Alcance de variables (local, global)
    4.3. Funciones de limpieza automatizada
5. Programación funcional con purrr
    5.1. Diferencias entre map(), map_dbl(), map_df()
    5.2. Uso de keep(), discard(), map_if()
    5.3. Iteraciones eficientes sobre listas y vectores
6. Diagnóstico y resumen de datos
    6.1. Uso de skimr::skim() para estadística descriptiva rápida

1. Repaso breve del curso básico

En esta sección repasaremos los conceptos esenciales del curso introductorio de R, que son la base para abordar los temas intermedios del curso actual.

1.1. Sintaxis general de R

```
# Asignación de variables
a <- 5
b <- 10
suma <- a + b
print(suma)
```

```
## [1] 15
```

```
# Comentarios se indican con el símbolo numeral
# Esto es un comentario

# Funciones básicas
sqrt(16)
```

```
## [1] 4
```

```
log(100, base = 10)
```

```
## [1] 2
```

1.2. Estructuras básicas

```
# Vectores
x <- c(1, 2, 3, 4, 5)

print(x)
```

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

```
# Matrices
m <- matrix(1:9, nrow = 3, byrow = TRUE)
print(m)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

```
# Listas
(mi_lista <- list(nombre = "Ana", edad = 25, notas = c(8, 9, 10)))
```

```
## $nombre
## [1] "Ana"
##
## $edad
## [1] 25
##
## $notas
## [1]  8  9 10
```

```
# Data frames
(df <- data.frame(nombre = c("Ana", "Luis"), edad = c(25, 30)))
```

```
##   nombre edad
## 1    Ana   25
## 2   Luis   30
```

1.3. Operadores lógicos

```
# Comparaciones
3 > 2       # TRUE
```

```
## [1] TRUE
```

```
5 == 5      # TRUE
```

```
## [1] TRUE
```

```
2 != 4      # TRUE
```

```
## [1] TRUE
```

```
# Operadores lógicos
TRUE & FALSE    # FALSE
```

```
## [1] FALSE
```

```
TRUE | FALSE    # TRUE
```

```
## [1] TRUE
```

1.4. Manejo de NaN, NULL y NA

```
# NA: valor faltante
y <- c(1, 2, NA, 4)
mean(y)             # NA
```

```
## [1] NA
```

```
mean(y, na.rm = TRUE)  # 2.33
```

```
## [1] 2.333333
```

```r
# NaN: resultado indefinido
z <- 0 / 0           # NaN

# NULL: objeto vacío
objeto_vacio <- NULL
length(objeto_vacio)  # 0
```

```
## [1] 0
```

Nota: En análisis estadístico es común encontrarse con datos faltantes o mal definidos, por lo que es importante conocer su tratamiento desde el inicio.

Instalar paquetes o bibliotecas

R cuenta con mucha variedad de paquetes, el más famoso `tidyverse` es una compilación de otros paquetes enfocados en distintas areas del análisis, visualización y manipulación de datos, entre otros.

Para usar un paquete de R, primero se tiene que instalar y después se tiene que activar:

```r
# install.packages("tidyverse")
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v ggplot2   3.5.2     v tibble    3.3.0
## v lubridate 1.9.4     v tidyr     1.3.1
## v purrr     1.0.4
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become error
```

Notar el uso de comillas en la primera función y en la segunda no.

2. Manejo avanzado de data frames

En R, los `data frames` son una de las estructuras de datos más utilizadas para trabajar con información tabular. Son similares a las hojas de cálculo de Excel o a tablas en bases de datos:

- Cada **columna** representa una variable.
- Cada **fila** representa una observación o registro.

En cursos básicos, normalmente aprendemos a:

- Crear un `data.frame`.
- Acceder a columnas.
- Hacer operaciones simples de filtrado.

Sin embargo, en **manejo avanzado** buscamos **eficiencia, flexibilidad y expresividad**, lo que significa que vamos a trabajar con:

1. **Indexación múltiple y condicional**
   - Acceder a filas y columnas específicas usando posiciones, nombres o condiciones lógicas.
   - Seleccionar datos cumpliendo múltiples criterios.
2. **Operaciones por columna y fila**
   - Calcular estadísticas por variable o por observación.
   - Crear nuevas columnas a partir de otras.
   - Aplicar funciones de manera eficiente con `apply()`, `rowSums()`, `rowMeans()` y funciones vectorizadas.
3. **Filtrado con múltiples condiciones y operadores**
   - Usar operadores lógicos (`&`, `|`, `!`) para combinar criterios.
   - Filtrar por valores específicos usando `%in%`.
   - Integrar librerías como `dplyr` para escribir código más legible y ordenado.

2.1. Indexación múltiple y condicional

En R podemos indexar un data.frame de varias maneras:

Por posición:

```
df <- data.frame(
  Nombre = c("Ana", "Luis", "María", "Pedro"),
  Edad = c(23, 31, 27, 40),
  Ciudad = c("CDMX", "Monterrey", "CDMX", "Guadalajara")
)

# Seleccionar la 1ª fila, 2ª columna
df[1, 2]
```

```
## [1] 23
```

```r
# Varias filas y columnas
df[1:3, c(1, 3)]
```

```
##   Nombre    Ciudad
## 1    Ana      CDMX
## 2   Luis Monterrey
## 3  María      CDMX
```

Por Nombre:

```r
# Columna por nombre
df[ , "Edad"]
```

```
## [1] 23 31 27 40
```

```r
# Varias columnas por nombre
df[ , c("Nombre", "Ciudad")]
```

```
##   Nombre      Ciudad
## 1    Ana        CDMX
## 2   Luis   Monterrey
## 3  María        CDMX
## 4  Pedro Guadalajara
```

por indexación condicional:

```r
# Personas con edad mayor a 30
df[df$Edad > 30, ]
```

```
##   Nombre Edad      Ciudad
## 2   Luis   31   Monterrey
## 4  Pedro   40 Guadalajara
```

```r
# Personas de CDMX con edad < 30
df[df$Ciudad == "CDMX" & df$Edad < 30, ]
```

```
##   Nombre Edad Ciudad
## 1    Ana   23   CDMX
## 3  María   27   CDMX
```

2.2. Operaciones por columna y fila

Operaciones vectorizadas por columna:

```r
# Nueva columna: Edad en meses
df$EdadMeses <- df$Edad * 12

# Promedio de edad
mean(df$Edad)
```

```
## [1] 30.25
```

Operaciones por fila con `rowSums()` y `rowMeans()`:

```r
notas <- data.frame(
  Alumno = c("A", "B", "C"),
  Parcial1 = c(80, 90, 75),
  Parcial2 = c(85, 88, 70),
  Parcial3 = c(90, 92, 80)
)

# Promedio por alumno
notas$Promedio <- rowMeans(notas[ , 2:4])

# Suma por alumno
notas$Suma <- rowSums(notas[ , 2:4])
```

Con la función `apply()`:

```r
# Suma por fila usando apply
apply(notas[ , 2:4], 1, sum)
```

```
## [1] 255 270 225
```

```r
# Máximo por columna
apply(notas[ , 2:4], 2, max)
```

```
## Parcial1 Parcial2 Parcial3
##       90       88       92
```

2.3. Filtrado con múltiples condiciones y operadores

Uso de operadores lógicos (And, Or, Not):

```r
# Alumnos con promedio > 85 y Parcial1 > 80
notas[notas$Promedio > 85 & notas$Parcial1 > 80, ]
```

```
##   Alumno Parcial1 Parcial2 Parcial3 Promedio Suma
## 2      B       90       88       92       90  270
```

```r
# Alumnos con promedio < 80 o Parcial3 < 80
notas[notas$Promedio < 80 | notas$Parcial3 < 80, ]
```

```
##   Alumno Parcial1 Parcial2 Parcial3 Promedio Suma
## 3      C       75       70       80       75  225
```

Uso de %in% para múltiples valores:

```r
# Filtrar ciudades específicas
df[df$Ciudad %in% c("CDMX", "Monterrey"), ]
```

```
##   Nombre Edad    Ciudad EdadMeses
## 1    Ana   23      CDMX       276
## 2   Luis   31 Monterrey       372
## 3  María   27      CDMX       324
```

Con `dyplr`:

```r
library(dplyr)

df %>%
  filter(Ciudad == "CDMX", Edad < 30) %>%
  select(Nombre, Edad)
```

```
##   Nombre Edad
## 1    Ana   23
## 2  María   27
```

```r
notas %>%
  filter(Promedio > 85 | Parcial3 > 85) %>%
  arrange(desc(Promedio))
```

```
##   Alumno Parcial1 Parcial2 Parcial3 Promedio Suma
## 1      B       90       88       92       90  270
## 2      A       80       85       90       85  255
```

3. Uso de dyplr

El paquete `dplyr` forma parte del conjunto de herramientas del `tidyverse` y está diseñado para manipular datos de manera clara, eficiente y reproducible.
Se basa en verbos que describen acciones sobre los datos, junto con el operador **pipe** (`%>%`) que permite encadenar operaciones.

3.1. Operaciones encadenas para el flujo de trabajo reproducible

En lugar de hacer múltiples asignaciones intermedias, podemos **encadenar pasos** para que el flujo de transformación de datos sea más legible.

```
library(dplyr)

# Ejemplo con mtcars: Filtrar autos con mpg > 20 y seleccionar variables clave
mtcars %>%
  filter(mpg > 20) %>%
  select(mpg, cyl, hp) %>%
  arrange(desc(mpg))
```

```
##                 mpg cyl  hp
## Toyota Corolla 33.9   4  65
## Fiat 128       32.4   4  66
## Honda Civic    30.4   4  52
## Lotus Europa   30.4   4 113
## Fiat X1-9      27.3   4  66
## Porsche 914-2  26.0   4  91
## Merc 240D      24.4   4  62
## Datsun 710     22.8   4  93
## Merc 230       22.8   4  95
## Toyota Corona  21.5   4  97
## Hornet 4 Drive 21.4   6 110
## Volvo 142E     21.4   4 109
## Mazda RX4      21.0   6 110
## Mazda RX4 Wag  21.0   6 110
```

Ventajas:

- Lectura secuencial del proceso.
- Código más corto y fácil de mantener.
- Menos objetos intermedios en memoria.

3.2. Verbos combinados: mutate(), group_by(), summarise(), arrange()

- mutate(): crea o transforma columnas.
- group_by(): agrupa datos por una o más variables.
- summarise(): resume información de cada grupo.
- arrange(): ordena filas.

```r
mtcars %>%
  mutate(consumo_kml = mpg * 0.425) %>%
  group_by(cyl) %>%
  summarise(
    promedio_hp = mean(hp),
    max_mpg = max(mpg),
    .groups = "drop"
  ) %>%
  arrange(desc(max_mpg))
```

```
## # A tibble: 3 x 3
##     cyl promedio_hp max_mpg
##   <dbl>       <dbl>   <dbl>
## 1     4        82.6    33.9
## 2     6       122.     21.4
## 3     8       209.     19.2
```

3.3. Limpieza de datos reales con filter() y case_when()

- filter(): selecciona filas según condiciones lógicas.
- case_when(): crea variables categóricas a partir de condiciones.

```r
df <- data.frame(
  nombre = c("Ana", "Luis", "María", "Pedro", "Juan"),
  edad = c(23, 31, 27, 40, 19),
  ciudad = c("CDMX", "Monterrey", "CDMX", "Guadalajara", "Monterrey")
)

df %>%
  filter(ciudad %in% c("CDMX", "Monterrey"), edad > 20) %>%
  mutate(
    grupo_edad = case_when(
      edad < 25 ~ "Joven",
      edad <= 35 ~ "Adulto joven",
      TRUE ~ "Adulto"
    )
  )
```

```
##   nombre edad    ciudad   grupo_edad
## 1    Ana   23      CDMX        Joven
## 2   Luis   31 Monterrey Adulto joven
## 3  María   27      CDMX Adulto joven
```

3.4. Uso de pivot_longer() y pivot_wider() desde tidyr

Estas funciones permiten cambiar el formato de los datos de ancho a largo y viceversa.

```r
library(tidyr)

# Ejemplo: formato ancho a largo
notas <- data.frame(
  alumno = c("A", "B", "C"),
  parcial1 = c(80, 90, 75),
  parcial2 = c(85, 88, 70),
```

```
  parcial3 = c(90, 92, 80)
)

# De ancho a largo
notas_largo <- notas %>%
  pivot_longer(
    cols = starts_with("parcial"),
    names_to = "examen",
    values_to = "calificacion"
  )

# De largo a ancho
notas_largo %>%
  pivot_wider(
    names_from = examen,
    values_from = calificacion
  )
```

```
## # A tibble: 3 x 4
##   alumno parcial1 parcial2 parcial3
##   <chr>     <dbl>    <dbl>    <dbl>
## 1 A            80       85       90
## 2 B            90       88       92
## 3 C            75       70       80
```

4. Introducción a funciones propias y programación modular

En R, las funciones nos permiten **automatizar tareas**, **reutilizar código** y mantener un **flujo de trabajo limpio y ordenado**.
La programación modular consiste en dividir un problema en piezas pequeñas (funciones) que se pueden desarrollar, probar y combinar fácilmente.

4.1. Sintaxis general de funciones

La estructura básica de una función en R es:

```
nombre_funcion <- function(argumento1, argumento2 = valor_por_defecto) {
  # Cuerpo de la función
  resultado <- argumento1 + argumento2
  return(resultado)
}

# Ejemplo de uso
suma_valores <- function(a, b) {
  return(a + b)
}

suma_valores(5, 3)
```

```
## [1] 8
```

Nota: Los argumentos pueden tener valores por defecto. A su vez, `return()` es opcional si la última línea de la función ya es el valor que queremos devolver.

4.2. Alcance de variables (local, global)

En R, las variables pueden tener alcance local (solo existen dentro de la función) o global (existen fuera y dentro de las funciones).

```
x <- 10  # Variable global

mi_funcion <- function() {
  x <- 5  # Variable local (solo dentro de la función)
  return(x)
}

mi_funcion()  # Devuelve 5 (local)
```

```
## [1] 5
```

```
x             # Sigue siendo 10 (global)
```

```
## [1] 10
```

Como regla general, si una variable se define dentro de una función, no afecta a la variable global con el mismo nombre. Y para modificar una variable global desde una función, se usa «- (pero se recomienda evitarlo salvo casos muy específicos).

4.3. Funciones de limpieza automatizada

Podemos crear funciones para automatizar transformaciones y limpieza de datos, evitando repetir código.

```
library(dplyr)

# Función para limpiar nombres y filtrar datos
limpiar_datos <- function(df, columna_filtrar, valor) {
  df %>%
    janitor::clean_names() %>%        # Limpia nombres de columnas
    filter(.data[[columna_filtrar]] == valor) %>%
    mutate(across(where(is.character), toupper))
}

# Ejemplo
df <- data.frame(
  Nombre = c("Ana", "Luis", "María"),
  Ciudad = c("CDMX", "Monterrey", "CDMX"),
  Edad = c(23, 31, 27)
)

limpiar_datos(df, "ciudad", "CDMX")
```

```
##   nombre ciudad edad
## 1    ANA   CDMX   23
## 2  MARÍA   CDMX   27
```

5. Programación funcional con purrr

El paquete **purrr** (parte de *tidyverse*) facilita aplicar funciones a cada elemento de un vector/lista (**mapear**) de forma **expresiva y tipada**, produciendo salidas consistentes (listas, dobles, data frames, etc.). Su sintaxis con fórmulas abreviadas (`~`) y pronombres (`.x`, `.y`) hace el código más conciso y reproducible.

5.1. Diferencias entre `map()`, `map_dbl()`, `map_df()`

- map() devuelve **lista** (salida genérica).
- map_dbl() obliga a que el resultado sea **numérico (double)**.
- map_df() (o map_dfr()) **apila filas** y devuelve un **data frame / tibble**.

```
library(purrr)
library(dplyr)
library(tibble)

# Ejemplo base: vector numérico
v <- c(2, 4, 6, 8)

# 1) map(): siempre lista
res_lista <- map(v, ~ .x^2)
str(res_lista)   # lista de 4 elementos
```

```
## List of 4
##  $ : num 4
##  $ : num 16
##  $ : num 36
##  $ : num 64
```

```
# 2) map_dbl(): vector numérico
res_dbl <- map_dbl(v, ~ .x^2)
res_dbl
```

```
## [1]  4 16 36 64
```

```
# 3) map_df(): devuelve tibble (una fila por elemento, o más si devuelves varios campos)
res_df <- map_df(v, ~ tibble(entrada = .x, cuadrado = .x^2))
res_df
```

```
## # A tibble: 4 x 2
##   entrada cuadrado
##     <dbl>    <dbl>
## 1       2        4
## 2       4       16
## 3       6       36
## 4       8       64
```

```
# Ejemplo con mtcars: calcular resumen por variable seleccionada
vars <- c("mpg", "hp", "wt")
map_df(vars, ~ tibble(var = .x,
                      media = mean(mtcars[[.x]]),
                      sd    = sd(mtcars[[.x]])))
```

```
## # A tibble: 3 x 3
##   var    media     sd
##   <chr>  <dbl>  <dbl>
## 1 mpg    20.1   6.03
## 2 hp    147.   68.6
## 3 wt      3.22  0.978
```

Cuándo usar cada una:

- Usa `map()` si la salida no es homogénea o necesitas una lista.
- Usa `map_dbl()` / `map_chr()` / `map_int()` cuando requieras un tipo específico.
- Usa `map_df()` (o `map_dfr()`) si quieres consolidar resultados "fila a fila" en un solo data frame.

5.2. Uso de keep(), discard(), map_if()

- `keep(.x, .p)` conserva elementos que cumplen el predicado `.p`.
- `discard(.x, .p)` descarta elementos que cumplen `.p`.
- `map_if(.x, .p, .f)` aplica `.f` solo a los elementos que cumplen `.p` (los demás se dejan igual).

```r
library(purrr)
library(dplyr)

# Lista heterogénea
L <- list(a = 1:3, b = "texto", c = 10:12, d = NA)

# 1) keep(): mantén solo los numéricos
solo_numericos <- keep(L, is.numeric)
solo_numericos
```

```
## $a
## [1] 1 2 3
##
## $c
## [1] 10 11 12
```

```r
# 2) discard(): elimina elementos vacíos o NA
sin_na <- discard(L, ~ all(is.na(.x)))
sin_na
```

```
## $a
## [1] 1 2 3
##
## $b
## [1] "texto"
##
## $c
## [1] 10 11 12
```

```r
# 3) map_if(): elevar al cuadrado solo los numéricos; lo demás intacto
L2 <- map_if(L, is.numeric, ~ .x^2)
L2
```

```
## $a
## [1] 1 4 9
##
## $b
## [1] "texto"
##
## $c
## [1] 100 121 144
##
## $d
## [1] NA
```

```
# Otro ejemplo: data frame y columnas numéricas
df <- as_tibble(mtcars)

# Estandarizar (z-score) únicamente columnas numéricas
df_std <- df %>%
  mutate(across(where(is.numeric), ~ (.-mean(.))/sd(.)))
df_std %>% select(mpg, hp, wt) %>% slice(1:3)
```

```
## # A tibble: 3 x 3
##     mpg     hp     wt
##   <dbl>  <dbl>  <dbl>
## 1 0.151 -0.535 -0.610
## 2 0.151 -0.535 -0.350
## 3 0.450 -0.783 -0.917
```

Tip: Para columnas, hoy se prefiere `dplyr::across()`; para listas o vectores arbitrarios, purrr es ideal.

5.3. Iteraciones eficientes sobre listas y vectores

purrr evita bucles explícitos y hace el código más declarativo. Además de `map()`, existen variantes para múltiples entradas:

- `map2(.x, .y, .f)`: itera en paralelo sobre dos entradas.
- `pmap(.l, .f)`: itera sobre lista de listas (o data frame por filas).
- `walk()`/`walk2()`/`pwalk()`: como map* pero para efectos secundarios (imprimir, escribir archivos).

```
library(purrr)
library(dplyr)
library(tidyr)
library(readr)

# 1) map() sobre archivos simulados ----------------------------------------
# (Ejemplo ilustrativo: crear 3 tibbles y "simular" que son archivos a leer)
set.seed(1)
tibbles <- map(1:3, ~ tibble(id = 1:5, valor = rnorm(5), archivo = paste0("file", .x)))
# Procesar cada tibble: filtrar y resumir
res_list <- map(tibbles, ~ .x %>% summarize(n = n(), media = mean(valor)))
bind_rows(res_list, .id = "origen")
```

```
## # A tibble: 3 x 3
##   origen     n  media
##   <chr>  <int>  <dbl>
```

```
## 1 1          5 0.129
## 2 2          5 0.135
## 3 3          5 0.0381
```

```
# 2) map2(): combinar dos vectores en paralelo ------------------------------
a <- 1:5
b <- 6:10
map2_dbl(a, b, ~ .x + .y)     # suma elemento a elemento
```

```
## [1]  7  9 11 13 15
```

```
# 3) pmap(): múltiples argumentos por fila ---------------------------------
param_grid <- tibble(mu = c(0, 1), sd = c(1, 2), n = c(5, 5))
# Para cada fila, generar n normales con (mu, sd) y devolver resumen
sim_res <- pmap(param_grid, function(mu, sd, n) {
  x <- rnorm(n, mean = mu, sd = sd)
  tibble(mu = mu, sd = sd, n = n, media = mean(x), min = min(x), max = max(x))
}) %>% bind_rows()
sim_res
```

```
## # A tibble: 2 x 6
##      mu    sd     n media     min   max
##   <dbl> <dbl> <dbl> <dbl>   <dbl> <dbl>
## 1     0     1     5 0.460 -0.0449 0.944
## 2     1     2     5 1.16  -2.98   2.84
```

```
# 4) walk(): efectos secundarios (no devuelve resultado útil) ---------------
temp_files <- paste0("tmp_", 1:3, ".txt")
walk2(temp_files, c("A","B","C"), ~ write_lines(.y, .x))  # escribir archivos
walk(temp_files, ~ message("Creado: ", .x))               # imprimir ruta
```

```
## Creado: tmp_1.txt
```

```
## Creado: tmp_2.txt
```

```
## Creado: tmp_3.txt
```

```
# Limpieza (opcional)
walk(temp_files, ~ if (file.exists(.x)) file.remove(.x))

# 5) Seguridad opcional: safely()/possibly() --------------------------------
# safely(): captura errores y devuelve list(result, error)
segura_log <- safely(log)
out <- map(list(1, 10, "x", -3), segura_log)
```

```
## Warning in .Primitive("log")(x, base): NaNs produced
```

```
out
```

```
## [[1]]
## [[1]]$result
## [1] 0
##
## [[1]]$error
```

```
## NULL
##
##
## [[2]]
## [[2]]$result
## [1] 2.302585
##
## [[2]]$error
## NULL
##
##
## [[3]]
## [[3]]$result
## NULL
##
## [[3]]$error
## <simpleError in .Primitive("log")(x, base): non-numeric argument to mathematical function>
##
##
## [[4]]
## [[4]]$result
## [1] NaN
##
## [[4]]$error
## NULL
```

```r
# possibly(): define valor por defecto en error
log_pos <- possibly(log, otherwise = NA_real_)
map_dbl(list(1, 10, "x", -3), ~ log_pos(.x))
```

```
## Warning in .Primitive("log")(x, base): NaNs produced
```

```
## [1] 0.000000 2.302585       NA      NaN
```

6. Diagnostico y resumen de datos

En esta sección usamos herramientas rápidas para: - Obtener **estadística descriptiva** clara y reproducible. - **Estandarizar nombres** de columnas para evitar errores y facilitar pipelines.

Recomendación: cargar estos paquetes al inicio del documento.

```
# install.packages(c("skimr", "janitor", "dplyr"))  # si hace falta
library(skimr)
library(janitor)
```

```
##
## Attaching package: 'janitor'
```

```
## The following objects are masked from 'package:stats':
##
##     chisq.test, fisher.test
```

```
library(dplyr)
```

6.1. Uso de skimr::skim() para estadística descriptiva rápida

skim() produce un resumen compacto por tipo de variable (numéricas, factores, fechas, etc.). Incluye conteos de NA, min, max, media, percentiles, y distribución (spark hist) cuando aplica.

Ejemplo con mtcars (convertimos algunas variables a factor para mostrar el comportamiento por tipo)

```
datos <- mtcars %>%
  mutate(
    cyl = factor(cyl),
    am  = factor(am, labels = c("Automática", "Manual"))
  )

# skim(datos)
```

Para documentos extensos, skim() puede ser verboso. Usa skim_without_charts() para evitar "sparklines":

```
skim_with(plots = list(histos = NULL))  # desactiva gráficos en skim (opcional)
```

```
## Creating new skimming functions for the following classes: histos.
## They did not have recognized defaults. Call get_default_skimmers() for more information.
```

```
## function (data, ..., .data_name = NULL)
## {
##     if (is.null(.data_name)) {
##         .data_name <- rlang::expr_label(substitute(data))
##     }
##     if (!inherits(data, "data.frame")) {
##         data <- as.data.frame(data)
##     }
##     stopifnot(inherits(data, "data.frame"))
##     selected <- names(tidyselect::eval_select(rlang::expr(c(...)),
```

```
##         data))
##     if (length(selected) == 0) {
##         selected <- names(data)
##     }
##     grps <- dplyr::groups(data)
##     if (length(grps) > 0) {
##         group_variables <- selected %in% as.character(grps)
##         selected <- selected[!group_variables]
##     }
##     else {
##         attr(data, "groups") <- list()
##     }
##     skimmers <- purrr::map(selected, get_final_skimmers, data,
##         local_skimmers, append)
##     types <- purrr::map_chr(skimmers, "skim_type")
##     unique_skimmers <- reduce_skimmers(skimmers, types)
##     combined_skimmers <- purrr::map(unique_skimmers, join_with_base,
##         base)
##     ready_to_skim <- tibble::tibble(skim_type = unique(types),
##         skimmers = purrr::map(combined_skimmers, mangle_names,
##             names(base$funs)), skim_variable = split(selected,
##             types)[unique(types)])
##     grouped <- dplyr::group_by(ready_to_skim, .data$skim_type)
##     nested <- dplyr::summarize(grouped, skimmed = purrr::map2(.data$skimmers,
##         .data$skim_variable, skim_by_type, data))
##     structure(tidyr::unnest(nested, "skimmed"), class = c("skim_df",
##         "tbl_df", "tbl", "data.frame"), data_rows = nrow(data),
##         data_cols = ncol(data), df_name = .data_name, dt_key = get_dt_key(data),
##         groups = dplyr::group_vars(data), base_skimmers = names(base$funs),
##         skimmers_used = get_skimmers_used(unique_skimmers))
## }
## <bytecode: 0x14d2723a8>
## <environment: 0x13cc645b0>
```

```
#skim(datos)
```

Puedes agrupar antes de resumir para comparar subpoblaciones:

```
# datos %>%
 # group_by(am) %>%
 # skim()
```

Si solo te interesan numéricas:

```
#datos %>%
 # select(where(is.numeric)) %>%
#skim()
```