Unidad IV

SESIÓN 1: Uso del paquete `tidyverse` en programación con R.

El manifiesto Tidyverse

El Manifiesto Tidyverse establece una serie de principios de diseño para lograr consistencia y coherencia en la interfaz de los paquetes del `tidyverse`, buscando que estos funcionen armónicamente. Es importante notar que se trata de un ideal aspiracional, y ningún paquete cumple actualmente con todos estos objetivos. Además, el manifiesto clarifica que los paquetes fuera del `tidyverse` no son inferiores, simplemente siguen enfoques de diseño distintos Link.

Los principios fundamentales de un paquete "tidy" son los siguientes:

- **Reutilizar las estructuras de datos existentes:** Se favorece el uso de estructuras de datos familiares como los `data frames` (particularmente los `tibbles`) para conjuntos de datos rectangulares, y los tipos de vectores base de R cuando sea posible.
- **Componer funciones simples con el operador pipe (%>%):** Se promueve la combinación de funciones sencillas y fáciles de entender mediante el operador pipe. Las funciones deben ser lo más simples posible, realizar una única tarea de manera eficiente, evitar mezclar efectos secundarios con transformaciones y tener nombres basados en verbos para mayor claridad.
- **Adoptar la Programación Funcional (PF):** El manifiesto alienta el uso de objetos inmutables, funciones genéricas provistas por S3 y S4, y herramientas que eviten los bucles `for` y `while` entre otros.
- **Diseñar para humanos:** *Se prioriza la facilidad de uso para los programadores por encima de la eficiencia computacional de la máquina.* Esto se logra mediante nombres de funciones evocadores y explícitos, así como el uso de prefijos comunes para las familias de funciones, facilitando la autocompletado y la comprensión.

```
# Activamos el paquete
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    3.2.1
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 becor
```

```
# Vemos el manifiesto en el IV cuadrante

vignette("manifesto", package = "tidyverse")
```

```
starting httpd help server ... done
```

Bloque de Inicialización de Librerías

Activamos los paquetes, en caso de no estarlo.

```r
library(tibble)
library(dplyr)
library(readr)
library(stringr)
library(purrr)
```

Aseguir revisamos los principales paquetes integrados en el ecosistema `tidyverse`.

`tibble`: La Base Inmutable de Datos

El paquete `tibble` se presenta como una evolución de los `data.frame` de R base, diseñado para ser más predecible y compatible con la filosofía de la programación funcional. Su característica más importante radica en su comportamiento **inmutable** en las operaciones del `tidyverse`. `tibble` actúa como la estructura de datos fundamental que esperan y devuelven las demás funciones del `tidyverse`, asegurando la consistencia en el flujo de trabajo.

Un ejemplo relevante para ilustrar su utilidad consiste en crear un `tibble` y demostrar cómo las transformaciones lo manipulan sin alterar su estado original.

```r
productos <- tibble(
  id = 1:3,
  nombre = c("Manzana", "Pera", "Uva"),
  precio = c(1.0, 1.2, 2.5)
)
productos # Tibble original de productos (inmutable)
```

```
# A tibble: 3 x 3
     id nombre  precio
  <int> <chr>    <dbl>
1     1 Manzana    1
2     2 Pera       1.2
3     3 Uva        2.5
```

```r
# Aplicamos una transformación para calcular el precio con IVA.
# mutate() es una función pura: para los mismos datos de entrada y la misma regla,
# siempre dará el mismo resultado. Además, no modifica 'productos'.
productos_con_iva <- productos %>%
  mutate(precio_iva = precio * 1.19)

productos_con_iva # Nuevo tibble con 'precio_iva' (resultado de una función pura)
```

```
# A tibble: 3 x 4
     id nombre  precio precio_iva
  <int> <chr>    <dbl>      <dbl>
1     1 Manzana    1         1.19
2     2 Pera       1.2       1.43
3     3 Uva        2.5       2.97
```

```
# Verificamos que el tibble original 'productos' NO ha cambiado.
# Esto demuestra el principio de INMUTABILIDAD.
productos # El tibble original 'productos' permanece INALTERADO
```

```
# A tibble: 3 x 3
     id nombre  precio
  <int> <chr>    <dbl>
1     1 Manzana    1
2     2 Pera       1.2
3     3 Uva        2.5
```

`tibble` se comporta como un bloque de construcción inmutable. Cuando se aplica una operación como `mutate()`, no se modifica el `tibble` original; en su lugar, se crea uno completamente nuevo con las transformaciones deseadas. Esta característica es fundamental para la predictibilidad y para evitar efectos secundarios no deseados en el código.

`readr`: Lectura Pura y Predecible

El paquete `readr` es el componente del `tidyverse` dedicado a la importación de datos. Sus funciones se caracterizan por ser **puras**, lo que significa que, para una misma ruta de archivo y las mismas opciones de configuración, siempre generarán el mismo `tibble` resultante sin sorpresas. `readr` complementa el ecosistema al proporcionar los `tibbles` iniciales (inmutables) con los que luego trabajarán `dplyr`, `stringr` y `purrr`.

Para ilustrar su utilidad, se presentará un ejemplo de cómo leer un archivo CSV de manera explícita, asegurando una carga de datos consistente y predecible.

```
# Preparamos una cadena de texto que simula el contenido de un archivo CSV simple.
csv_simple <- "id,nombre,valor
1,ItemA,100
2,ItemB,150
3,ItemC,200"

# Creamos un archivo temporal para simular la lectura de un archivo real.
```

```r
temp_csv_file <- tempfile(fileext = ".csv")
write_file(csv_simple, temp_csv_file)

# read_csv() es una función PURA: dado el mismo archivo, siempre
# devolverá el mismo tibble. No tiene efectos secundarios.
datos_cargados <- read_csv(
  temp_csv_file,
  col_types = cols( # Explicitamos los tipos de columna para mayor predictibilidad
    id = col_integer(),
    nombre = col_character(),
    valor = col_integer()
  )
)
datos_cargados # Datos cargados con read_csv()
```

```
# A tibble: 3 x 3
     id nombre valor
  <int> <chr>  <int>
1     1 ItemA    100
2     2 ItemB    150
3     3 ItemC    200
```

```r
class(datos_cargados) # Verificando la clase del objeto cargado (siempre un tibble por defec
```

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

```r
# Eliminamos el archivo temporal.
unlink(temp_csv_file)

# Al especificar col_types, la lectura es determinista.
# Si se leyera el mismo archivo con los mismos argumentos múltiples veces,
# el 'tibble' resultante sería idéntico, sin variaciones por inferencia automática.
```

`readr` otorga la garantía de que los datos se cargarán siempre de la misma manera. Se lo puede concebir como una "máquina de hacer `tibbles`" que produce consistentemente el mismo resultado dados los mismos insumos, lo que previene errores sutiles derivados de la inferencia automática de tipos de datos.

```r
# Bloque de demostración: Ejemplos de funciones menos "puras" o predecibles

# --- Demostración de read.csv() (R base) ---
# Problema: Comportamiento inconsistente de la salida (tipo de objeto)
# read.csv() no siempre devuelve un data.frame. Si lee una sola columna, puede devolver un ve
# Esto rompe la predictibilidad y la facilidad de composición en pipes.

# Simular un CSV con una sola columna
writeLines("columna_unica\n10\n20\n30", "single_column.csv")

# Comportamiento cuando read.csv lee una sola columna
datos_columna_unica_base <- read.csv("single_column.csv")
cat("Clase de la salida de read.csv (una columna): ")
print(class(datos_columna_unica_base)) # Puede ser "data.frame"
cat("Clase de la columna accedida: ")
print(class(datos_columna_unica_base$columna_unica)) # Generalmente "numeric" (un vector)
print(datos_columna_unica_base)
cat("\n")

# Comparación con readr::read_csv() para consistencia
# (Se asume que library(readr) ya fue llamado en un bloque inicial)
datos_columna_unica_readr <- readr::read_csv("single_column.csv", show_col_types = FALSE)
cat("Clase de la salida de readr::read_csv() (una columna): ")
print(class(datos_columna_unica_readr)) # Siempre "tbl_df", "tbl", "data.frame" (un tibble)
print(datos_columna_unica_readr)
cat("\n")

# Limpiar archivo temporal
unlink("single_column.csv")


# Problema: Inferencia automática de 'stringsAsFactors' (histórico, pre-R 4.0)
# Antes de R 4.0, read.csv convertía cadenas a factores por defecto,
# un efecto secundario en la interpretación del tipo de dato.
writeLines("id,nombre,tipo\n1,Manzana,Fruta\n2,Pera,Fruta\n3,Leche,Lacteo", "data_factors.csv

cat("--- Demostración de stringsAsFactors en read.csv() (histórico/forzado) ---\n")
datos_con_factores <- read.csv("data_factors.csv", stringsAsFactors = TRUE)
cat("Clase de la columna 'tipo' con stringsAsFactors = TRUE: ")
print(class(datos_con_factores$tipo)) # Muestra "factor"
print(datos_con_factores)
cat("\n")
```

```r
# Limpiar archivo temporal
unlink("data_factors.csv")

# --- Demostración de funciones con efectos secundarios globales ---
# Problema: Modifican el estado global del entorno de R, afectando la predictibilidad
# de otras operaciones y la reproducibilidad del script.

cat("--- Demostración de setwd() (efecto secundario global) ---\n")
# Guardamos el directorio de trabajo actual para poder restaurarlo después
original_wd <- getwd()
cat("Directorio de trabajo original: ", original_wd, "\n")

# Este es un efecto secundario: modifica el entorno global
setwd(tempdir()) # Cambia el directorio de trabajo a uno temporal
cat("Nuevo directorio de trabajo (efecto secundario): ", getwd(), "\n")

# Cualquier operación posterior que use rutas relativas se verá afectada.
# Por ejemplo, si intentaras leer un archivo que estaba en el WD original, fallaría.
# tryCatch(read.csv("archivo_que_no_existe_en_temp.csv"), error = function(e) message("Error
```

`dplyr`: La Maquinaria de Transformación Funcional

`dplyr` se presenta como el núcleo del `tidyverse` para la manipulación de datos. Sus funciones (`filter()`, `mutate()`, `group_by()`, `summarise()`, `select()`, etc.) son inherentemente **puras** y están diseñadas para la **composición** fluida mediante el operador pipe (`%>%`), siempre respetando la **inmutabilidad** de los datos. `dplyr` complementa a los otros paquetes al tomar los `tibbles` iniciales de `readr` y transformarlos en nuevos `tibbles` de forma legible y segura, listos para ser utilizados por `stringr` o procesados por `purrr`.

Un ejemplo de utilidad relevante se centrará en el cálculo de totales simples por categoría, mostrando múltiples pasos de composición.

```r
transacciones <- tibble(
  categoria = c("A", "B", "A", "C", "B"),
  monto = c(100, 50, 120, 200, 75)
)
transacciones # Datos de transacciones originales
```

```
# A tibble: 5 x 2
  categoria monto
  <chr>     <dbl>
```

```
1 A           100
2 B            50
3 A           120
4 C           200
5 B            75
```

```r
# Cada paso devuelve un NUEVO tibble, manteniendo la INMUTABILIDAD
resumen_transacciones <- transacciones %>%
  # Paso 1: Agrupar por categoría
  # group_by() es PURA: define grupos lógicos, no modifica los datos en sí.
  group_by(categoria) %>%
  # Paso 2: Sumar el monto total por cada categoría
  # summarise() es PURA: reduce cada grupo a una fila de resumen, sin efectos secundarios.
  summarise(total_monto = sum(monto)) %>%
  # Paso 3: Filtrar categorías con un monto total superior a 100
  # filter() es PURA: selecciona filas según una condición, sin efectos secundarios.
  filter(total_monto > 100) %>%
  # Paso 4: Desagrupar el tibble para futuras operaciones
  # ungroup() es PURA: remueve la estructura de agrupación.
  ungroup()

resumen_transacciones # Resumen de transacciones procesado (nuevo tibble, original inalterado
```

```
# A tibble: 3 x 2
  categoria total_monto
  <chr>           <dbl>
1 A                 220
2 B                 125
3 C                 200
```

```r
transacciones # Verificando que los datos originales NO se han modificado
```

```
# A tibble: 5 x 2
  categoria monto
  <chr>     <dbl>
1 A           100
2 B            50
3 A           120
4 C           200
5 B            75
```

`dplyr` funciona como una "fábrica modular" donde cada "estación" (`group_by`, `summarise`, `filter`) recibe un producto (el `tibble`), realiza una operación pura y pasa el resultado a la siguiente estación. El resultado final es un nuevo producto sin haber alterado el original, lo que simplifica enormemente la construcción y comprensión de flujos de análisis complejos.

`stringr`: Manipulación de Texto Pura y Consistente

`stringr` se presenta como una herramienta que simplifica la manipulación de cadenas de texto en R, ofreciendo funciones **puras** con una sintaxis notablemente consistente. Este paquete complementa el ecosistema al permitir la limpieza y estandarización de datos textuales dentro de los `tibbles` (comúnmente utilizando `dplyr::mutate`), ya sea antes o después de otras transformaciones.

Un ejemplo relevante para ilustrar su utilidad es la estandarización de nombres simples.

```
usuarios <- tibble(
  id = 1:3,
  nombre_usuario = c("  ALICE  ", "bOb", " charlie ")
)
usuarios # Nombres de usuario originales
```

```
# A tibble: 3 x 2
     id nombre_usuario
  <int> <chr>
1     1 "  ALICE  "
2     2 "bOb"
3     3 " charlie "
```

```
# Limpieza y estandarización de nombres usando stringr dentro de dplyr::mutate
# str_trim, str_to_lower, str_to_title son funciones PURAS.
usuarios_limpios <- usuarios %>%
  mutate(
    nombre_limpio = str_trim(nombre_usuario),       # Elimina espacios al inicio/final
    nombre_estandar = str_to_title(str_to_lower(nombre_limpio)) # Pasa a minúsculas y luego a
  )

usuarios_limpios # Nombres de usuario estandarizados (nuevo tibble, original inalterado)
```

```
# A tibble: 3 x 4
     id nombre_usuario nombre_limpio nombre_estandar
  <int> <chr>          <chr>         <chr>
```

```
1     1 "  ALICE  "    ALICE         Alice
2     2 "bOb"          bOb           Bob
3     3 " charlie "    charlie       Charlie
```

```
usuarios # El tibble original 'usuarios' no ha cambiado (INMUTABILIDAD)
```

```
# A tibble: 3 x 2
     id nombre_usuario
  <int> <chr>
1     1 "  ALICE  "
2     2 "bOb"
3     3 " charlie "
```

Cuando se trabaja con datos textuales, que a menudo son "sucios", `stringr` proporciona herramientas limpias y puras para su procesamiento. Cada función realiza su tarea específica sin afectar otros elementos, lo que facilita el encadenamiento de operaciones complejas de texto de manera predecible.

Tratamiento de acentos y otros caractere

```
# Ejemplo simple con stringr para caracteres especiales

# Datos sucios con acentos y tildes/virgulillas
nombres_sucios <- c("José", "Mañana", "Niño~")

nombres_limpios <- nombres_sucios %>%
  str_to_lower() %>% # 1. Convertir a minúsculas para uniformidad
  str_replace_all("~", "") %>% # 2. Transliterar acentos a ASCII (aproximación)
  iconv(from = "UTF-8", to = "ASCII//TRANSLIT") # 3. Eliminar el carácter '~'

print(nombres_limpios)
```

```
[1] "jose"   "manana" "nino"
```

`purrr`: Iteración Funcional de Alto Nivel

`purrr` reemplaza los bucles (como for y while) explícitos con funciones que operan sobre colecciones de datos (listas y vectores) de manera funcional. Sus funciones de alto orden promueven la **composición** y la **pureza** en las iteraciones. `purrr` complementa el `tidyverse` al permitir aplicar secuencias de transformaciones a múltiples `tibbles` (o cualquier otro objeto) de forma consistente y reproducible.

Composición de Operaciones con el Operador Pipe (%>%) en `purrr`

El operador pipe (%>%) es fundamental en **purrr** para componer una secuencia de operaciones sobre colecciones de datos de forma legible y encadenada. En ciencia de datos, esto es común para flujos de limpieza y transformación paso a paso.

```
# Un vector de datos de calidad de aire con posibles entradas inconsistentes
datos_calidad_aire_crudos <- c("25.5", "18.2", "40.1*", "15.0", "30.7?")

# Aplicamos una serie de transformaciones encadenadas para limpiar y convertir los datos
datos_calidad_aire_limpios <- datos_calidad_aire_crudos %>%
  str_replace_all("[^0-9.]", "") %>% # Eliminar caracteres no numéricos
  as.numeric() %>%                   # Convertir a número
  map(~ .x * 1.05) %>%               # Ajustar por un factor de calibración del 5%
  flatten_dbl()                      # Convertir la lista resultante en un vector numérico

datos_calidad_aire_limpios # Datos de calidad de aire después de la limpieza y ajuste
```

```
[1] 26.775 19.110 42.105 15.750 32.235
```

```
datos_calidad_aire_crudos # Los datos crudos originales no han sido modificados
```

```
[1] "25.5"  "18.2"  "40.1*" "15.0"  "30.7?"
```

El operador pipe permite una **composición secuencial** clara de múltiples pasos. Cada función toma la salida del paso anterior, manteniendo la pureza e inmutabilidad. Esto facilita la lectura y el mantenimiento del código.

Iteración Pura con `map()`

La familia de funciones `map` en **purrr** es clave para realizar iteraciones funcionales. Permite aplicar una función a cada elemento de una lista o vector, devolviendo una nueva lista (o vector de tipo específico) sin modificar el original. Es especialmente útil para procesar conjuntos de datos o modelos de forma individual pero consistente.

```
# Lista de data frames, simulando diferentes muestras de datos de pacientes

muestras_pacientes <- list(
  tibble(id = 1, edad = 25, peso = 70),
  tibble(id = 2, edad = 40, peso = 85),
  tibble(id = 3, edad = 30, peso = 60)
)
muestras_pacientes # Muestras de pacientes originales
```

```
[[1]]
# A tibble: 1 x 3
     id  edad  peso
  <dbl> <dbl> <dbl>
1     1    25    70

[[2]]
# A tibble: 1 x 3
     id  edad  peso
  <dbl> <dbl> <dbl>
1     2    40    85

[[3]]
# A tibble: 1 x 3
     id  edad  peso
  <dbl> <dbl> <dbl>
1     3    30    60
```

```r
# Definimos una función pura para calcular el IMC para un data frame de paciente
# Esta función es PURA: mismo resultado para misma entrada, sin efectos secundarios.
calcular_imc <- function(df_paciente) {
  df_paciente %>%
    mutate(imc = peso / ((edad / 100)^2)) # Simplificado, solo para demostración
}

# map() aplica 'calcular_imc' a cada data frame de 'muestras_pacientes'.
# El resultado es una NUEVA lista de data frames, manteniendo la INMUTABILIDAD.
muestras_pacientes_con_imc <- map(muestras_pacientes, calcular_imc)

muestras_pacientes_con_imc # Muestras de pacientes con IMC calculado (Iteración Funcional)
```

```
[[1]]
# A tibble: 1 x 4
     id  edad  peso   imc
  <dbl> <dbl> <dbl> <dbl>
1     1    25    70  1120

[[2]]
# A tibble: 1 x 4
     id  edad  peso   imc
  <dbl> <dbl> <dbl> <dbl>
1     2    40    85  531.
```

```
[[3]]
# A tibble: 1 x 4
     id  edad  peso   imc
  <dbl> <dbl> <dbl> <dbl>
1     3    30    60  667.
```

```
muestras_pacientes # La lista original no ha sido modificada
```

```
[[1]]
# A tibble: 1 x 3
     id  edad  peso
  <dbl> <dbl> <dbl>
1     1    25    70

[[2]]
# A tibble: 1 x 3
     id  edad  peso
  <dbl> <dbl> <dbl>
1     2    40    85

[[3]]
# A tibble: 1 x 3
     id  edad  peso
  <dbl> <dbl> <dbl>
1     3    30    60
```

`map()` permite componer una operación de procesamiento sobre múltiples elementos sin un bucle `for` explícito, resultando en un código más declarativo y menos propenso a errores. Esto es invaluable en ciencia de datos para procesamiento por lotes, aplicación de modelos a subconjuntos de datos o limpieza de listas de data frames. El resultado es un código más limpio y escalable.

Complemento: Principios de Programación Funcional en Tidyverse

- La **Programación Funcional** (PF) es un paradigma que construye programas mediante la composición de funciones puras e inmutables, tratando las funciones como valores para lograr código predecible y escalable.

En este apartado analizamos los principios de programación funcional (PF) —funciones puras, inmutabilidad y composición de funciones— en el contexto del paquete **tidyverse** de R, dirigido a un público de nivel magister. Cada principio se define, se ilustra con un ejemplo

en tidyverse y se contrasta con un caso que lo viola. Se incluye una tabla comparativa para sintetizar las ideas.

Funciones Puras

Definición: Una función pura produce el mismo resultado para los mismos argumentos y no genera efectos secundarios (e.g., modificar estados externos). Esto garantiza predictibilidad y facilidad de razonamiento.

Ejemplo en Tidyverse:

```
library(tidyverse)
numeros <- list(1, 2, 3, 4)
map_dbl(numeros, ~ .x^2) # Resultado: [1, 4, 9, 16]
```

```
[1]  1  4  9 16
```

La función `~ .x^2` es pura, ya que siempre produce `.x^2` para cada entrada `.x` sin alterar el entorno.

Contraejemplo:

```
contador <- 0
no_pura <- function(x) {
  contador <<- contador + 1
  x + contador
}
no_pura(5) # Resultado: 6 (primera vez)
```

```
[1] 6
```

```
no_pura(5) # Resultado: 7 (segunda vez)
```

```
[1] 7
```

La función `no_pura` viola la pureza al depender de y modificar `contador`, un estado externo.

Inmutabilidad

Definición: La inmutabilidad implica no modificar datos originales, generando copias con los cambios aplicados. Esto evita perdida del dataset original y asegura consistencia en eluso de datos.

Ejemplo en Tidyverse:

```
library(tidyverse)
datos <- tibble(x = c(1, 2, 3))
nuevo <- datos %>% mutate(x_doble = x * 2)
nuevo # x = c(1, 2, 3), x_doble = c(2, 4, 6)
```

```
# A tibble: 3 x 2
      x x_doble
  <dbl>   <dbl>
1     1       2
2     2       4
3     3       6
```

```
datos # Original sin cambios
```

```
# A tibble: 3 x 1
      x
  <dbl>
1     1
2     2
3     3
```

`mutate()` respeta la inmutabilidad al devolver un nuevo tibble sin alterar `datos`.

Contraejemplo:

```
datos <- tibble(x = c(1, 2, 3))
datos$x[1] <- 10 # Modifica directamente
datos # x = c(10, 2, 3)
```

```
# A tibble: 3 x 1
      x
  <dbl>
1    10
2     2
3     3
```

Modificar directamente `datos` viola la inmutabilidad al alterar el objeto original.

Composición de Funciones

Definición: La composición de funciones encadena funciones pequeñas, donde la salida de una es la entrada de la siguiente, promoviendo un estilo declarativo y modular.

Ejemplo en Tidyverse:

```
datos <- tibble(nombre = c("Ana", "Bob", "Cris"), edad = c(25, 30, 35))
result <- datos %>%
  filter(edad > 25) %>%
  mutate(edad_doble = edad * 2) %>%
  summarise(edad_promedio = mean(edad_doble))
result # edad_promedio = 65
```

```
# A tibble: 1 x 1
  edad_promedio
          <dbl>
1            65
```

El operador `%>%` compone `filter()`, `mutate()` y `summarise()` de forma clara e inmutable.

Contraejemplo:

```
datos <- tibble(x = c(1, 2, 3))
resultado <- numeric()
for (i in 1:nrow(datos)) {
  if (datos$x[i] > 1) {
    resultado <- c(resultado, datos$x[i] * 2)
  }
}
resultado # c(4, 6)
```

```
[1] 4 6
```

El bucle imperativo gestiona el flujo manualmente, perdiendo la modularidad de la composición funcional.

Tabla Comparativa

Principio	Descripción	Ejemplo en Tidyverse	Contraejemplo
Funciones Puras	Mismo resultado para mismos argumentos, sin efectos secundarios.	`map_dbl(list(1, 2, 3, 4), ~ .x^2)`	`no_pura(5)` modifica `contador`
Inmutabilidad	No modificar datos originales; crear copias con cambios.	`datos %>% mutate(x_doble = x * 2)`	`datos$x[1] <- 10` modifica directamente
Composición de Funciones	Encadenar funciones, salida de una como entrada de otra.	`datos %>% filter(...) %>% mutate(...) %>% summarise(...)`	Bucle `for` para filtrar y transformar datos

El paquete `tidymodels`

Modelos inferenciales: Un ANOVA de un factor

```
library(tidymodels)
```

```
-- Attaching packages ------------------------------------- tidymodels 1.3.0 --

v broom        1.0.8     v rsample      1.3.0
v dials        1.4.0     v tune         1.3.0
v infer        1.0.8     v workflows    1.2.0
v modeldata    1.4.0     v workflowsets 1.1.0
v parsnip      1.3.1     v yardstick    1.3.2
v recipes      1.3.1

-- Conflicts ---------------------------------------- tidymodels_conflicts() --
x scales::discard() masks purrr::discard()
x dplyr::filter()   masks stats::filter()
x recipes::fixed()  masks stringr::fixed()
x dplyr::lag()      masks stats::lag()
x yardstick::spec() masks readr::spec()
x recipes::step()   masks stats::step()
```

```r
library(rlang)
```

```
Adjuntando el paquete: 'rlang'

The following objects are masked from 'package:purrr':

    %@%, flatten, flatten_chr, flatten_dbl, flatten_int, flatten_lgl,
    flatten_raw, invoke, splice
```

```r
# --- Ejemplo 1: ANOVA (Regresión Lineal con Factor) con tidymodels ---
# Dataset: mtcars
# Objetivo: Determinar si existe una diferencia en mpg (millas por galón)
#           entre diferentes números de cilindros (cyl).
# Output deseado: La tabla ANOVA indica si hay diferencia de grupos.

# 1. Preparación de Datos (Funcional e Inmutable)
data_mtcars_anova_tidy <- mtcars %>%
  as_tibble() %>%
  mutate(cyl = factor(cyl))

# 2. Especificación del Modelo (Funcional)
modelo_spec_anova <- linear_reg() %>%
  set_engine("lm")

# 3. Ajuste del Modelo (Funcional)
modelo_fit_anova <- modelo_spec_anova %>%
  fit(mpg ~ cyl, data = data_mtcars_anova_tidy)

# 4. Extracción de Resultados para ANOVA (Funcional) - ¡CORRECCIÓN AQUÍ!
# Para obtener la tabla ANOVA de un objeto 'model_fit' de tidymodels:
# Accedemos al modelo subyacente ('fit') y luego usamos 'anova()',
# seguido de 'tidy()' de broom para un tibble limpio y funcional.

resumen_anova_tidy <- modelo_fit_anova %>%
  pluck("fit") %>% # Extrae el objeto 'lm' subyacente del objeto model_fit
  anova() %>%      # Aplica la función anova() de R base al objeto 'lm'
  tidy()           # Convierte la tabla ANOVA resultante en un tibble

# Mostrar la tabla ANOVA
# La columna 'p.value' para 'cyl' nos dice si hay diferencia significativa.
```

```r
# Un p.value < 0.05 sugiere que SÍ hay diferencia.
print("Tabla ANOVA (para ver si existe diferencia entre grupos):")
```

```
[1] "Tabla ANOVA (para ver si existe diferencia entre grupos):"
```

```r
print(resumen_anova_tidy)
```

```
# A tibble: 2 x 6
  term         df sumsq meansq statistic  p.value
  <chr>     <int> <dbl>  <dbl>     <dbl>    <dbl>
1 cyl           2  825.  412.       39.7  4.98e-9
2 Residuals    29  301.   10.4      NA   NA
```

Modelo de clasificador: variables significativas

```r
# --- Ejemplo 2: Modelo Logístico Lineal con tidymodels (usando Titanic) ---

library(titanic)

# Queremos predecir si un pasajero del Titanic sobrevivió (Survived)
# usando su clase (Pclass), sexo (Sex) y si era niño (Child).

# 1. Preparar datos: Usar titanic_train, limpiar NAs y crear variables.
datos_titanic <- titanic_train %>%
  as_tibble() %>%
  select(Survived, Pclass, Sex, Age) %>%
  drop_na() %>% # Quitar filas con datos faltantes (en 'Age')
  mutate(
    Survived = factor(Survived, levels = c(0, 1), labels = c("No", "Yes")),
    Pclass = factor(Pclass),
    Sex = factor(Sex),
    Child = factor(ifelse(Age < 18, "Yes", "No"))
  ) %>%
  select(-Age) # Quitar la edad numérica que ya no necesitamos

# 2. Definir el modelo: Regresión logística que usará 'glm' de R base.
```

```r
modelo_logistico_def <- logistic_reg() %>%
  set_engine("glm")

# 3. Ajustar el modelo: Entrenar el modelo con los datos.
modelo_logistico_ajustado <- modelo_logistico_def %>%
  fit(Survived ~ Pclass + Sex + Child, data = datos_titanic)

# 4. Ver resultados: Extraer los coeficientes y sus p-valores en formato limpio.
tabla_coeficientes_logistico <- modelo_logistico_ajustado %>%
  tidy() # Limpiar los coeficientes del modelo

# Muestra la tabla de coeficientes. Mira la columna 'p.value'.
# Si p.value es bajo (ej. < 0.05) para una variable, es significativa.
print("Tabla de Coeficientes Logísticos (¿qué variables son significativas?):")
```

```
[1] "Tabla de Coeficientes Logísticos (¿qué variables son significativas?):"
```

```r
print(tabla_coeficientes_logistico)
```

```
# A tibble: 5 x 5
  term        estimate std.error statistic  p.value
  <chr>          <dbl>     <dbl>     <dbl>    <dbl>
1 (Intercept)     2.25     0.240      9.40 5.58e-21
2 Pclass2        -1.01     0.260     -3.88 1.05e- 4
3 Pclass3        -2.21     0.251     -8.80 1.33e-18
4 Sexmale        -2.53     0.206    -12.3  9.85e-35
5 ChildYes        1.07     0.271      3.95 7.91e- 5
```

SESIÓN 2: Modelos descriptivos y predictivos con `tidymodels`

Modelos descriptivos

Random forest e inferencia Bayesiana en `tidymodels`

Referencias

- Mailund, T. (2017). Functional programming in R: Advanced statistical programming for data science, analysis and finance (1ª ed.). Apress.