Sesión 5

Programación funcional en R

Campamento de invierno EGOB | UC | 04 de agosto, 2023

José D. Conejeros | jdconejeros@uc.cl

Guía

1. Programación funcional
2. Estructuras de control
3. Funciones
4. Familia `apply`
5. Paquete `purrr`

A nivel específico

EL objetivo de esta clase es comprender los fundamentos de la programación funcional para su uso en la gestión de grandes volúmenes de información.

Estructuras de control

- Introducción
- Comando `if`
- Comando `else`
- Comando `ifelse`
- Encadenar `if` y `else`
- Comando `case_when`
- Comando `for`
- Comando `while`
- Otras estructuras de control

Funciones

- Idea general de función
- Creación de funciones: `function()`
- Ejemplos

Familia `apply`

- Función `apply`
- Función `tapply`
- Función `lapply`
- Función `sapply`

Paquete `purrr`

- Introducción
- Función `map` y variantes
- Función `map2` y `pmap`
- Función `walk` y `nest` y otras

Programación funcional

R, en esencia, es un lenguaje funcional. En otras palabras, su estilo de resolución de problemas está centrado en funciones.

- Funciones de primera clase: puede trabajar una función como si fuera un vector. Puede asignarlas a variables, almacenarlas en listas, pasarlas como argumentos de otras funciones, crearlas dentro de funciones y incluso devolverlos como resultado de una función.

- Funciones puras:

    - La salida solo depende de las entradas, es decir, si lo vuelve a llamar con las mismas entradas, obtendrá las mismas salidas.
    - La función no tiene efectos secundarios, como cambiar el valor de una variable global.

Sin embargo, R no requiere de la escritura de funciones puras $\rightarrow$ R tiene un **estilo funcional**.

La programación funcional es complementaria a la programación orientada a objetos, que ha sido el paradigma de programación dominante durante las últimas décadas.

Estructuras de control

En programación, una estructura de control es un tipo de comando que permite controlar cuándo se ejecuta un código, mediante sentencias condicionales, las cuales se definen usando operadores lógicos:

Lógicos: permiten realizar preguntas, que serán respondidas como verdadero (TRUE) o falso (FALSE)

- `&` → Y (Intersección)
- `|` → O (Unión)
- `!` → Negación
- `~` → Negación
- `%in%` → Dentro de

Relacionales: nos permiten evaluar condiciones.

- `>` → Mayor que
- `<` → Menor que
- $\geqslant$ → Mayor o igual que
- $\leqslant$ → Menor o igual que
- $=$ → Igual (doble igual)
- $\neq$ → No igual
- `~=` → No igual

Estructuras de control

Ejemplos de uso de operadores lógicos

```
10 > 5
```

```
[1] TRUE
```

```
"UDP" = "udp"
```

```
c("a", "B", "c") %in% letters
```

```
[1]  TRUE FALSE  TRUE
```

Estructuras de control

En programación, una estructura de control es un tipo de comando que permite controlar cuándo se ejecuta un código, **mediante sentencias condicionales**:

- Es un número mayor o igual a 4:

```
10 ⩾ 4
```

```
[1] TRUE
```

- Es un número par:

```
10 %% 2 == 0
```

```
[1] TRUE
```

Las estructuras de control más utilizadas son:

- `if/else`: ejecuta código a partir de una condición
- `for`: bucle que se repite n veces
- `while`: bucle que se ejecuta mientras una condición es verdadera
- `repeat`: repite un bucle indefinidamente
- `break`: detiene un bucle
- `next`: salta a la siguiente ejecución de un bucle

Estructuras de control: `if`

Se usa para ejecutar un código sólo cuando se cumpla una condición específica. Se usa de la siguiente forma:

```
if(condicion){
  Ejecuta una acción si la condición es VERDADERA
}
```

Por ejemplo:

```
nota_tarea ← 5
if(nota_tarea ⩾ 4){
  print("¡Felicitaciones!")
}
```

```
[1] "¡Felicitaciones!"
```

```
nota_tarea ← 3
if(nota_tarea ⩾ 4){
  print("¡Felicitaciones!")
}
```

Estructuras de control: `else`

Se usa para añadir condiciones a un if. Esto se usa de la siguiente forma:

```
if(condicion){
  Ejecuta una acción si la condición es VERDADERA
} else{
  Ejecuta una acción si la condición es FALSA
}
```

Continuemos con el ejemplo:

```
nota_tarea ← 3
if(nota_tarea ⩾ 4){
  print("¡Felicitaciones!")
} else {
  "Reprobaste"
}
```

```
[1] "Reprobaste"
```

Estructuras de control: `ifelse`

Une las dos anteriores en una sola función. Se usa de la siguiente forma:

```
ifelse(Condicion,
       Ejecuta una acción si la condición es VERDADERA,
       Ejecuta una acción si la condición es FALSA
       )
```

Veamos el ejemplo anterior

```
nota_tarea ← 3
if(nota_tarea ⩾ 4){
  print("¡Felicitaciones!")
} else {
  "Reprobaste"
}
```

```
[1] "Reprobaste"
```

```
nota_tarea ← 3
ifelse(nota_tarea ⩾ 4,
       "¡Felicitaciones!",
       "Reprobaste")
```

```
[1] "Reprobaste"
```

Estructuras de control: `else` y `if`

Veamos el siguiente ejemplo:

```
nota_tarea ← 8
ifelse(nota_tarea ⩾ 4,
       "¡Felicitaciones!",
       "Reprobaste")
```

```
[1] "¡Felicitaciones!"
```

Se pueden encadenar `if` y `else` para generar condicionales más robustos. Con esto podemos corregir el error anterior:

```
nota_tarea ← 8
if((nota_tarea < 1) | (nota_tarea > 7) | (is.numeric(nota_tarea) == FALSE)) {
print("Error, ingrese un número entre 1 y 7")
} else if (nota_tarea ⩾ 4) {
print("¡Felicitaciones!")
} else {
print("Reprobaste")
}
```

```
[1] "Error, ingrese un número entre 1 y 7"
```

Estructuras de control: anidar `ifelse`

Con la idea de anidación podemos resolver el problema anterior.

```
ifelse((nota_tarea < 1) | (nota_tarea > 7) | (is.numeric(nota_tarea) == FALSE),
       "Error, ingrese un número entre 1 y 7",
       ifelse(nota_tarea ≥ 4,
              "¡Felicitaciones!",
              "Reprobaste"))
```

```
[1] "Error, ingrese un número entre 1 y 7"
```

La librería de `dplyr` tiene la función `if_else()` que se aplica de la misma forma.

Estructuras de control: anidar `case_when`

Función de la librería `dplyr` que permite evaluar múltiples condiciones `if` e `ifelse` de forma simultanea (solo actúa cuando la condición es verdadera). En otras palabras, **vectoriza** múltiples `if_else()`. Funciona de la siguiente manera:

```
case_when(condicion1 ~ <Resultado si condición 1 es TRUE,
          condicion2 ~ <Resultado si condición 2 es TRUE,
          condicion3 ~ <Resultado si condición 3 es TRUE,
           ...
          TRUE ~ NA)
```

Por ejemplo:

```
case_when(nota_tarea < 1 ~ "Error, ingrese un número entre 1 y 7.",
          nota_tarea > 7 ~ "Error, ingrese un número entre 1 y 7.",
          is.numeric(nota_tarea) == FALSE ~ "Error, ingrese un número.",
          nota_tarea >= 4 ~ "¡Felicitaciones!",
          nota_tarea < 4 ~ "Reprobaste")
```

```
[1] "Error, ingrese un número entre 1 y 7."
```

Esta función es particularmente útil cuando trabajamos con variables categóricas.

Estructuras de control: iteraciones `for`

Ejecuta un código para todos los elementos indicados en un bucle (*loop*). Se utiliza en conjunto a otras estructuras para generar funciones. Opera de la siguiente manera:

```
for(x in vector_a_recorrer){
  Código a ejecutar en cada caso, en términos de "x"
}
```

x es auxiliar y que va **iterando** de elemento en elemento. Veamos un ejemplo:

```
for(i in 1:5){
  print(i^2 + i)
}
```

```
[1] 2
[1] 6
[1] 12
[1] 20
[1] 30
```

Para cada elemento en el objeto, ejecute la siguiente sentencia.

Estructuras de control: iteraciones `for`

Veamos el siguiente ejemplo:

```
x ← 1:5
y ← c()
for(i in 1:5){
y[i] ← x[i]^2
}
y
```

```
[1]  1  4  9 16 25
```

Este resultado es equivalente a vectorizar:

```
x ← 1:5
y ← x^2
y
```

```
[1]  1  4  9 16 25
```

En R hay opciones más simples y óptimas que aplicar bucles de tipo `for`: `apply`, `map`, entre otras.

Estructuras de control: anidar `for`

Podemos anidar:

```
x ← matrix(1:6, 2, 3)
x
```

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

```
for(i in seq_len(nrow(x))) {
  for(j in seq_len(ncol(x))) {
    print(x[i, j])
  }
  }
```

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

Estructuras de control: `while`

Los bucles `while` comienzan probando una condición. Si es cierto, ejecutan el cuerpo del bucle. Una vez que se ejecuta el cuerpo del bucle, la condición se prueba de nuevo y así sucesivamente. Se usa de la siguiente forma:

```
white(condicion es VERDADERA){
  codigo que se ejecuta
}
```

Ejemplo:

```
count ← 0
while(count < 5){
  count ← count + 1
  print(count)
  }
```

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

Otras estructuras de control: `break`, `next` y `repeat`

Se utilizan para modificar el comportamiento de funciones iterativas (`while` y `for`)

- `break`: Permite terminar la ejecución de código en bucle
- `next`: Permite saltar la ejecución de una interacción en un bucle

El comando `repeat` permite repetir una secuencia de comandos. Es necesario definir un break para que esta iteración pare, en otro caso, seguirá indefinidamente. Funciona de la siguiente manera:

```
repeat {
  codigo a ejecutar
  break para quebrar el codigo
}
```

Otras estructuras de control: `break`, `next` y `repeat`

Veamos un ejemplo:

```
i = 1
repeat {
  print(i)
  i ← i+1
  if (i>5) break
  }
```

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

Otras estructuras de control: `break`, `next` y `repeat`

Apliquemos `break` y `next`:

```
for(i in 1:10) {
  if(i == 3) {
    break
    }
  print(i)
  }
```

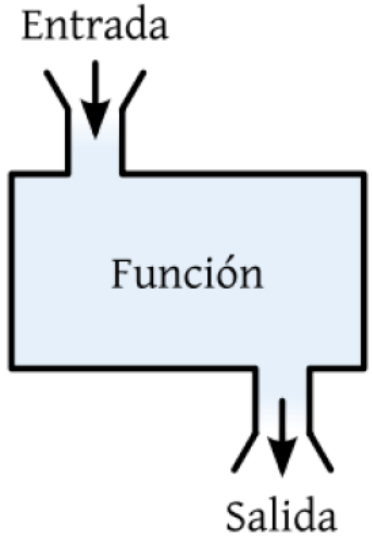
```
[1] 1
[1] 2
```

```
for(i in 1:10) {
  if(i == 3) {
    next
    }
  print(i)
  }
```

```
[1] 1
[1] 2
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

Funciones

Dentro de la programación se pueden generar funciones propias para ciertos procesos, ya que permite optimizar el código y el trabajo futuro. Podemos realizar funciones de la siguiente manera:




```
nombre_funcion = function(arg1, arg2, ... ) {
  Cuerpo de la función, cálculos en término de los argumentos
  return(devuelve un resultado)
}
```

Funciones: ejemplos

- Función que calcula $x^2 + x$:

```
# Construímos la función
fun1 = function(x) {
  x^2 + x
}
# Aplicamos
fun1(1)
```

```
[1] 2
```

```
fun1(1:10)
```

```
 [1]   2   6  12  20  30  42  56  72  90 110
```

¿Cuándo escribir funciones? Cuando has copiado y pegado un bloque de código más de dos veces.

Funciones: ejemplos

- Función que calcula $ax^2 + bx + c$:

```
# Construímos la función
fun2 = function(x, a, b, c) {
  a*x^2 + b*x + c
  }
# Aplicamos
fun2(x=1, a=10, b=5, c=1)
```

```
[1] 16
```

```
fun2(x=1:10, a=10, b=5, c=1)
```

```
 [1]   16   51  106  181  276  391  526  681  856 1051
```

Funciones:

- El valor de retorno de una función es la última expresión en el cuerpo de la función que se evaluará.
- Las funciones se pueden pasar como argumentos a otras funciones.
- Las funciones se pueden anidar, de modo que pueda definir una función dentro de otra función.
- La función `formals()` devuelve una lista de todos los argumentos formales de una función.

```
formals(fun2)
```

```
$x
```

```
$a
```

```
$b
```

```
$c
```

Funciones:

Explique la siguiente función:

```
fun3 ← function(x){
  a ← mean(x, na.rm = TRUE)
  b ← sd(x, na.rm = TRUE)
  x ← (x-a)/b
  return(head(x))
}
```

Apliquemos nuestra función con datos reales:

```
data ← datos::paises
fun3(x=paises$esperanza_de_vida)
```

```
[1] -2.374637 -2.256112 -2.127213 -1.970599 -1.810501 -1.628572
```

Las funciones que creamos nosotros permanecen en el ambiente de R temporariamente. Cuando removemos los objetos del ambiente, la función deja de existir. Por ende, debemos incorporarla en cada uno de los scripts en la cual la necesitemos. Una buena práctica, es incorporar nuestras funciones útiles al comienzo de cada script junto a la carga de las librerías.

Funciones:

Podemos aplicar los condicionales y procesos iterativos vistos previamente.

```
fun4 ← function(x, y, z) {
  if (z < 0.1) {
    sum(x, y)
    } else {
      sum(x, y) * 1.1
    }
  }

fun4(x=10, y=1, z=0.05)
```

```
[1] 11
```

```
fun4(x=10, y=1, z=1)
```

```
[1] 12.1
```

Aplicación de funciones: familia `apply`

Son una familia de funciones que permiten realizar una acción repetitivamente en múltiples fragmentos de datos (matrices, data frames, arrays y listas). Son esencialmente bucles, pero se ejecuta más rápido que los bucles y, a menudo, requiere menos código.

- `apply()`
- `eapply()`
- `lapply()`
- `mapply()`
- `rapply()`
- `sapply()`
- `tapply()`
- `vapply()`

Con esta familia de funciones automatizamos tareas complejas usando poca líneas de código y es una de las características distintivas de R como lenguaje de programación.

Función `apply`

Se aplica una función `FUN` a una **matriz** `X`.

- Se puede aplicar a las FILAS: `MARGIN=1`
- Se puede aplicar a las COLUMNAS: `MARGIN=2`

Veamos un ejemplo:

```
apply(X=data[,4:6], MARGIN=2, FUN = mean, na.rm=TRUE)
```

```
esperanza_de_vida         poblacion    pib_per_capita
     5.947444e+01      2.960121e+07      7.215327e+03
```

```
apply(X=data[,4:6], MARGIN=1, FUN = mean, na.rm=TRUE) %>% head()
```

```
[1] 2808714 3080595 3422656 3846279 4360079 4960399
```

Función `tapply`

Se aplica una función `FUN` a un **objeto** `X` segregando por el objeto INDEX.

```
tapply(X=data$esperanza_de_vida, INDEX=data$continente, FUN = mean, na.rm=TRUE)
```

```
  África Américas     Asia   Europa  Oceanía 
48.86533 64.65874 60.06490 71.90369 74.32621 
```

Función `lapply`

Se aplica una función `FUN` a una **lista** `X`. Retorna un objeto en formato lista.

```
Lista = list("Esperanza de vida" = data$esperanza_de_vida,
             "Población" = data$poblacion,
             "PIB" = data$pib_per_capita)

lapply(X = Lista, FUN = mean, na.rm=TRUE)
```

```
$`Esperanza de vida`
[1] 59.47444

$Población
[1] 29601212

$PIB
[1] 7215.327
```

Función `sapply`

Se aplica una función `FUN` a una **lista** `X`. Retorna un objeto en formato vector.

```
Lista = list("Esperanza de vida" = data$esperanza_de_vida,
             "Población" = data$poblacion,
             "PIB" = data$pib_per_capita)

sapply(X = Lista, FUN = mean, na.rm=TRUE)
```

```
Esperanza de vida         Población               PIB
     5.947444e+01      2.960121e+07      7.215327e+03
```

Función apply vs for

Comparemos ambos procedimientos:

```
# Aplicando apply
inicio ← Sys.time() # Tiempo de inicio
apply(X=data[,4:6], MARGIN=2, FUN = mean,
```

```
esperanza_de_vida          poblacion    pib_per_
     5.947444e+01       2.960121e+07      7.2153
```

```
fin ← Sys.time() # Tiempo al final
fin - inicio # Tiempo de ejecución
```

```
Time difference of 0.002104998 secs
```

```
# Mediante un for
inicio ← Sys.time()
variables ← colnames(data[,4:6])
for(i in variables){
  print(mean(data[[i]], na.rm=TRUE))
}
```

```
[1] 59.47444
[1] 29601212
[1] 7215.327
```

```
fin ← Sys.time()
fin - inicio
```

```
Time difference of 0.005585909 secs
```

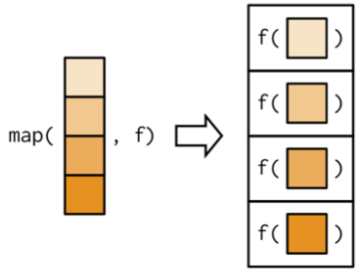
Paquete purrr

El paquete purrr contiene múltiples funciones para trabajar con vectores/listas y funciones, esto con el objetivo de mejorar la manera en que se aplican funciones a vectores, listas, etc. Lo anterior, permite ahorrar posibles procesos con iteraciones, transformándolo en un código más limpio y de fácil lectura.

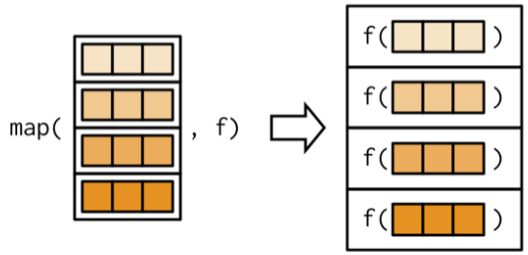
Este paquete es fundamental para la **programación funcional**, ya que busca una programación limpia basada en la aplicación y composición de funciones. Se busca que el cálculo se base en funciones puras, estas son funciones que siempre retornarán el mismo resultado, sin posibilidad de que elementos externos o internos modifiquen su funcionamiento.

Familia `map`

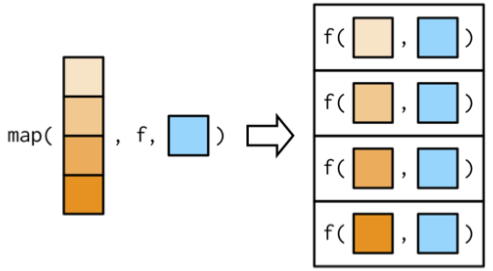
Es el equivalente al `apply`. Permite aplicar funciones a elementos de listas o vectores pero está optimizado al ambiente `tidyverse`.



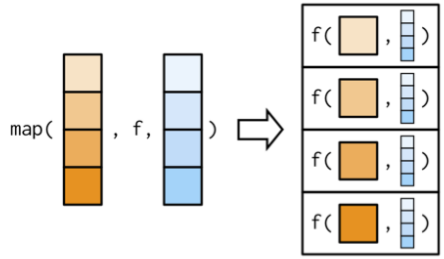
Aplicado a un vector



Aplicado a una lista



Aplicado a un vector y a un elemento



Aplicado a dos vectores

Las funciones map() transforman su entrada aplicando una función a cada elemento de una lista o vector y devolviendo un objeto de la misma longitud que la entrada.

Función map()

Permite aplicar funciones a los elementos de vectores/listas. La función map siempre retorna una lista.

```
Lista = list("Esperanza de vida" = data$esperanza_de_vida,
             "Población" = data$poblacion,
             "PIB" = data$pib_per_capita)

purrr::map(Lista, mean, na.rm=TRUE)
```

```
$`Esperanza de vida`
[1] 59.47444

$Población
[1] 29601212

$PIB
[1] 7215.327
```

Comparemos `map` vs `sapply`

```
inicio ← Sys.time()
sapply(X = Lista, FUN = mean, na.rm=TRUE)
```

```
Esperanza de vida         Población               PIB
     5.947444e+01      2.960121e+07      7.215327e+03
```

```
fin ← Sys.time()
fin - inicio
```

```
Time difference of 0.001811981 secs
```

Comparemos `map` vs `sapply`

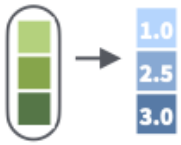
```
inicio ← Sys.time()
purrr::map(Lista, mean, na.rm=TRUE)
```

```
$`Esperanza de vida`
[1] 59.47444

$Población
[1] 29601212

$PIB
[1] 7215.327
```

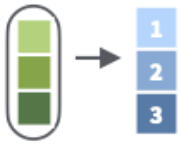
```
fin ← Sys.time()
fin - inicio
```

```
Time difference of 0.002017975 secs
```

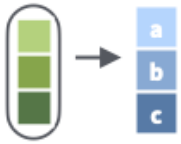
Funciones derivadas de los `map`



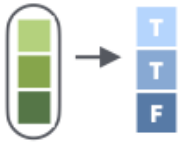
map_dbl(.x, .f, …**)**
Return a double vector.
map_dbl(x, mean)



map_int(.x, .f, ...**)**
Return an integer vector.
map_int(x, length)



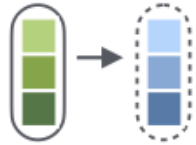
map_chr(.x, .f, …**)**
Return a character vector.
map_chr(l1, paste, collapse = "")



map_lgl(.x, .f, …**)**
Return a logical vector.
map_lgl(x, is.integer)

map_dfc(.x, .f, ...**)**
Return a data frame created by column-binding.
map_dfc(l1, rep, 3)

map_dfr(.x, .f, ..., .id = NULL**)**
Return a data frame created by row-binding.
map_dfr(x, summary)



walk(.x, .f, ...**)** Trigger side effects, return invisibly.
walk(x, print)

Funciones derivadas de los `map`

`map2`: Aplica una función a pares de elementos a partir de dos listas/vectores. Entrega una lista.

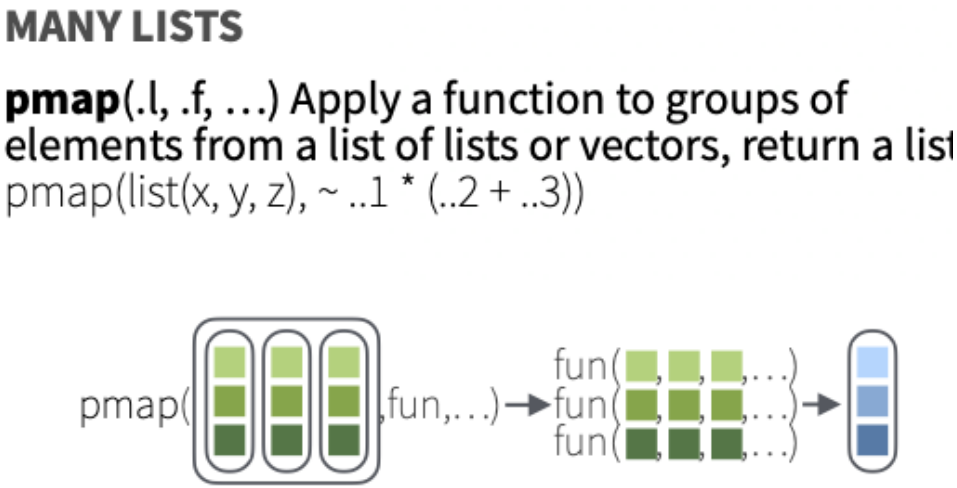
TWO LISTS

map2(.x, .y, .f, …**)** Apply a function to pairs of elements from two lists or vectors, return a list.

```
y <- list(1, 2, 3); z <- list(4, 5, 6); l2 <- list(x = "a", y = "z")
map2(x, y, ~ .x * .y)
```

Funciones derivadas de los `map`

`pmap`: Es una generalización de map2 a un número indeterminado p de argumentos enlistados. Entrega una lista



Estas funciones también tienen alternativas que permiten modificar el tipo de resultado obtenido anteriormente, añadiendo alguno de los sufijos `_chr`, `_dbl`, `_dfc`, etc.

Función map2 y pmap

Veamos un ejemplo:

```
# Definimos 3 listas
Lista1 = list(data$continente)
Lista2 = list(data$anio)
Lista3 = list(data$pais)
```

```
purrr::map2(Lista1, Lista2, table)
```

```
[[1]]

          1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 2002 2007
  África    52   52   52   52   52   52   52   52   52   52   52   52
  Américas  25   25   25   25   25   25   25   25   25   25   25   25
  Asia      33   33   33   33   33   33   33   33   33   33   33   33
  Europa    30   30   30   30   30   30   30   30   30   30   30   30
  Oceanía    2    2    2    2    2    2    2    2    2    2    2    2
```

Función map2 y pmap

Veamos un ejemplo:

```
# Definimos una lista de valores
l ← list(Lista3, Lista2, Lista1)
purrr::pmap(l, table)
```

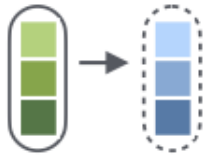
```
[[1]]
, ,  = África
```

	1952	1957	1962	1967	1972	1977	1982	1987	1992
Afganistán	0	0	0	0	0	0	0	0	0
Albania	0	0	0	0	0	0	0	0	0
Argelia	1	1	1	1	1	1	1	1	1
Angola	1	1	1	1	1	1	1	1	1
Argentina	0	0	0	0	0	0	0	0	0
Australia	0	0	0	0	0	0	0	0	0
Austria	0	0	0	0	0	0	0	0	0
Baréin	0	0	0	0	0	0	0	0	0
Bangladesh	0	0	0	0	0	0	0	0	0
Bélgica	0	0	0	0	0	0	0	0	0
Benin	1	1	1	1	1	1	1	1	1
Bolivia	0	0	0	0	0	0	0	0	0

Otras herramientas de `purrr`

Funciones `walk()`

Las funciones `walk()`, `walk2()` y `pwalk()` son muy similares a las funciones map. Sin embargo, estas son utilizadas cuando se desea aplicar una acción en vez de generación. Algunos ejemplos de acciones son: `save()`, `ggsave()`, `write_csv()`, `print()`, etc.



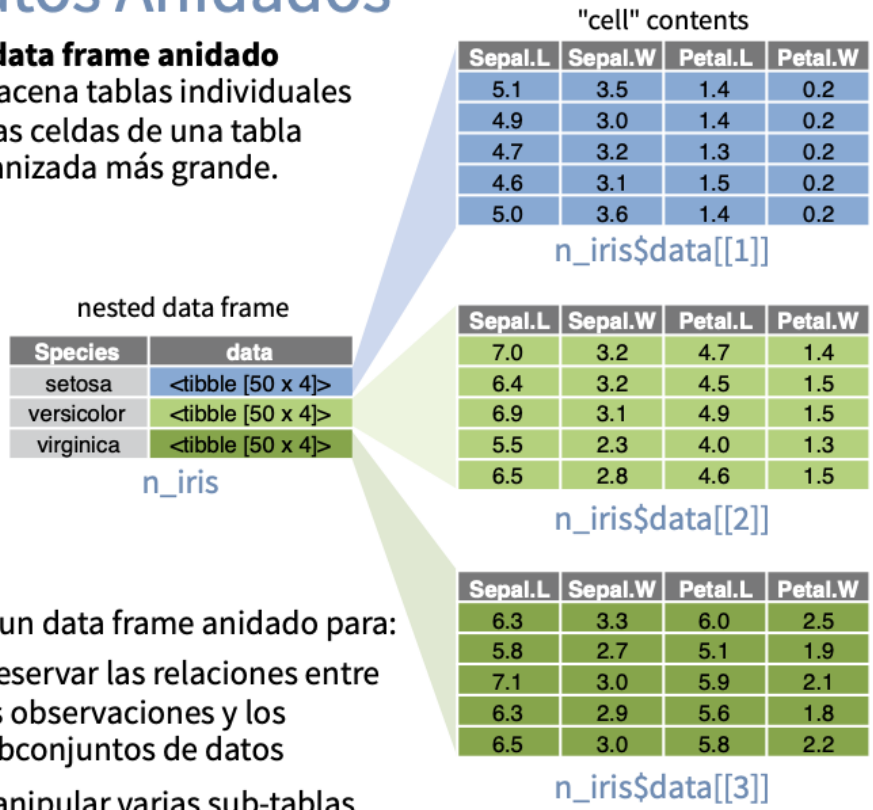
walk(.x, .f, ...**)** Trigger side effects, return invisibly.
walk(x, print)

Otras herramientas de `purrr`

Funciones `nest()`

Datos Anidados

Un **data frame anidado** almacena tablas individuales en las celdas de una tabla organizada más grande.



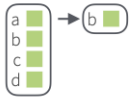
Usa un data frame anidado para:

- Preservar las relaciones entre las observaciones y los subconjuntos de datos
- manipular varias sub-tablas a la vez con las funciones **purrr** **map()**, **map2()**, o **pmap()**.

Otras herramientas de purrr

Trabajar con listas

FILTRAR LISTAS

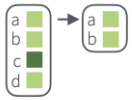


pluck(.x, ..., .default=NULL) Selecciona un elemento por nombre o índice, *pluck(x,"b")* ,o su atributo con **attr_getter**. *pluck(x,"b",attr_getter("n"))*

keep(.x, .p, …) Selecciona elementos que pasan una prueba lógica. *keep(x, is.na)*

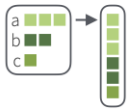
discard(.x, .p, …) Selecciona elementos que no pasan una prueba lógica. *discard(x, is.na)*

compact(.x, .p = identity) Elimina elementos vacíos. *compact(x)*

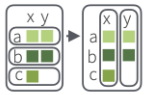


head_while(.x, .p, …) Devuelve elementos desde el principio hasta que uno no pasa. También **tail_while**. *head_while(x, is.character)*

REMODELAR LISTAS



flatten(.x) Elimina un nivel de índices de una lista. También **flatten_chr**, **flatten_dbl**, **flatten_dfc**, **flatten_dfr**, **flatten_int**, **flatten_lgl**. *flatten(x)*



transpose(.l, .names = NULL) Transpone el orden del índice en una multi-lista. *transpose(x)*

RESUMIR LISTAS

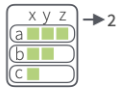
every(.x, .p, …) ¿Pasan todos los elementos una prueba? *every(x, is.character)*

some(.x, .p, …) ¿Pasan algunos elementos una prueba? *some(x, is.character)*

has_element(.x, .y) ¿Contiene la lista un elemento? *has_element(x, "foo")*

detect(.x, .f, ..., .right=FALSE, .p) Encuentra el primer elemento que pasa. *detect(x, is.character)*

detect_index(.x, .f, ..., .right = FALSE, .p) Encuentra el índice del primer elemento que pasa. *detect_index(x, is.character)*



depth(x) Devuelve profundidad (número de niveles o índices). *depth(x)*

UNIR LISTAS

append(x, values, after = length(x)) Añade al final de una lista. *append(x, list(d = 1))*

prepend(x, values, before = 1) Añade al principio de una lista. *prepend(x, list(d = 1))*

splice(…) Combina objetos en una lista, almacena objetos S3 como sub-listas. *splice(x, y, "foo")*

TRANSFORMAR LISTAS

modify(.x, .f, ...) Aplica una función a cada elemento. También **map**, **map_chr**, **map_dbl**, **map_dfc**, **map_dfr**, **map_int**, **map_lgl**. *modify(x, ~.+2)*

modify_at(.x, .at, .f, ...) Aplica una función a los elementos por nombre o índice. También **map_at**. *modify_at(x, "b", ~.+2)*

modify_if(.x, .p, .f, ...) Aplica una función a los elementos que pasan una prueba. También **map_if**. *modify_if(x, is.numeric,~.+2)*

modify_depth(.x,.depth,.f,...) Aplica una función a cada elemento y un nivel dado de una lista. *modify_depth(x, 1, ~.+2)*

TRABAJAR CON LISTAS

array_tree(array, margin = NULL) Convierte una matriz en una lista. También **array_branch**. *array_tree(x, margin = 3)*

cross2(.x, .y, .filter = NULL) Todas las combinaciones de .x e .y. También **cross**, **cross3**, **cross_df**. *cross2(1:3, 4:6)*

set_names(x, nm = x) Fija el nombre de un vector/lista directamente o con una función. *set_names(x, c("p", "q", "r"))* *set_names(x, tolower)*

Otras herramientas de purrr: Ejemplo

¿Qué hace esta función?

```
dir.create("Bases_Continentes")
data %>%
  group_nest(continente) %>%
  mutate(file = paste0("Bases_Continentes/", continente, ".csv"))
```

```
# A tibble: 5 × 3
  continente               data file
  <fct>      <list<tibble[,5]>> <chr>
1 África              [624 × 5] Bases_Continentes/África.csv
2 Américas            [300 × 5] Bases_Continentes/Américas.csv
3 Asia                [396 × 5] Bases_Continentes/Asia.csv
4 Europa              [360 × 5] Bases_Continentes/Europa.csv
5 Oceanía              [24 × 5] Bases_Continentes/Oceanía.csv
```

Otras herramientas de `purrr`: Ejemplo

¿Qué hace esta función?

```
dir.create("Bases_Continentes")
data %>%
  group_nest(continente) %>%
  mutate(file = paste0("Bases_Continentes/", continente, ".csv")) %>%
  select(file, x = data) %>%
  pwalk(write_csv)
```

```
# Veamos e resultado
list.files("Bases_Continentes/")
```

```
[1] "África.csv"   "Américas.csv" "Asia.csv"     "Europa.csv"   "Oceanía.csv"
```

Referencias y material complementario

Wickham, H., & Grolemund, G. (2016). R for data science: import, tidy, transform, visualize, and model data. " O'Reilly Media, Inc.". Cap. 18 al 21. Recurso en línea: https://es.r4ds.hadley.nz/

Wickham, H. (2019). Advanced r. CRC press. Cap. 9 al 11. Recurso en línea: https://adv-r.hadley.nz/fp.html

Otras referencias de interés

- Página web del paquete purrr (en inglés)
- Cheatsheet de purrr (en inglés)
- Cheatsheet de R Avanzado (en inglés)

Sesión 5

Programación funcional en R

04 de agosto, 2023

José D. Conejeros | jdconejeros@uc.cl | JDConejeros